

# INFORMIX-ESQL/C

Version 4.00

Embedded SQL and Tools for C

Programmer's Manual





## Reader Response Card

Dear Reader,

At Informix Software, we think documentation is very important. After all, manuals are an integral part of our product line. Because documentation *is* important, we want to know what you think about our manuals. You can tell us by filling out and returning this Reader Response Card.

Thanks for your help.

1. What is the title of your manual and the version number of the product?

---

2. Overall, how do you rate this manual?

- ☐ Outstanding    ☐ Very Good    ☐ Good  
☐ Average    ☐ Poor

What did you like most about it?

---



---

What would it take to make it better?

---



---

3. Is the manual effective?

- Is it organized so you can find things? ☐ Yes ☐ No
- Is the index adequate? ☐ Yes ☐ No
- Is the table of contents easy to use? ☐ Yes ☐ No
- If applicable, are the chapter summaries useful? ☐ Yes ☐ No
- If applicable, do you like the way we show statement syntax? ☐ Yes ☐ No
- Are topics presented in a useful sequence? ☐ Yes ☐ No

4. What did you think of the writing?

- Overall writing quality:  
☐ Outstanding    ☐ Very Good    ☐ Good    ☐ Average    ☐ Poor
- Clarity:  
☐ Very clear    ☐ Average    ☐ Hard to understand
- Tone:  
☐ Friendly    ☐ Neutral    ☐ Stilted    ☐ Patronizing
- Level:  
☐ Just right    ☐ Oversimplified    ☐ Too technical

5. What is your job title or area of responsibility?

---

6. What Informix products do you use?

---

Additional Comments:

---

---

---

Your Name 

---

Company Name 

---

Address 

---

---

City 

---

 State or Province 

---

Zip or Postal Code 

---

 Country 

---

To mail this card, please fold on the dotted line and staple or tape the open end. No postage is necessary if mailed in the United States.

Informix is a registered trademark of Informix Software, Inc.

000-10059-22



NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES

---

## BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 25 MENLO PARK, CA.

POSTAGE WILL BE PAID BY ADDRESSEE



Informix Software, Inc.  
Technical Publications Department  
4100 Bohannon Drive  
Menlo Park, California 94025



## **New Features in ESQL/C Version 4.0**

A number of enhancements have been made to **INFORMIX-ESQL/C** Version 4.0. Listed here are the new features that are available on both the **INFORMIX-SE** and **INFORMIX-OnLine** database engines. Enhancements that require the **INFORMIX-OnLine** database engine are described in the **INFORMIX-OnLine** documentation.

### **ANSI Level I and II SQL Modifications**

The additions and modifications made to **INFORMIX-ESQL/C** place it in full compliance with the ANSI Level I SQL standards and partial compliance with the ANSI Level II SQL standards.

The changes affect many of the existing SQL statements and include the following features:

**Data type synonyms**    The following data type synonyms are supported:

- **REAL** for **SMALLFLOAT**
- **DOUBLE PRECISION** for **FLOAT**
- **CHARACTER** for **CHAR**
- **NUMERIC** and **DEC** for **DECIMAL**
- **INT** for **INTEGER**

**UNIQUE and  
DISTINCT**

The keywords **UNIQUE** and **DISTINCT** are now synonyms.

**NULL values**

**NULL** values are considered identical when evaluated within a **GROUP BY** clause. Previously, each **NULL** value was treated as a separate group.

- Comment indicators** You can now use a double dash (--) to indicate a comment in an INFORMIX-ESQL/C program.
- INDICATOR keyword** The syntax of indicator variables has been extended to allow the INDICATOR keyword.
- Non-short indicators** Indicator variables no longer have to be declared as short integers. They can now be any data type except datetime, interval, or locator.

## MODE ANSI Databases

You now have the option of specifying a database as MODE ANSI when you CREATE or START it. In a MODE ANSI database, transactions are implicit, and you are always within a transaction. The name of an object is qualified by the owner of the object, where *owner* is the login name of the user that created the object.

## Checking for ANSI Compatibility

You can define the DBANSIWARN environment variable to check for Informix extensions to ANSI standard syntax at compile time or run time, or use the -ansi flag at compile time for this purpose.

The sqlwarn flags, already used to report a variety of warning conditions, also are set to indicate conditions related to ANSI compatibility.

## New Data Types and Keywords

Two modifications made to INFORMIX-ESQL/C are proposed for inclusion in SQL2:

- The new data types DATETIME and INTERVAL are supported.
- You can use the UNIQUE CONSTRAINT keywords in the CREATE TABLE statement to require that a single column or composite column accept only unique data.



## **New Sort Package**

A new sort package speeds index generation and the execution of queries that include an ORDER BY clause or the SELECT UNIQUE keywords.

## **New Query Optimizer**

A new query optimizer can dramatically improve the speed of queries performed on large tables. In addition, you can use the new syntax SET EXPLAIN ON/OFF to view the access path that the optimizer takes in executing your query.

## **New Utilities**

INFORMIX-ESQL/C provides two new utilities, **dbimport** and **dbexport**, for importing and exporting database data.

## **New Preprocessor**

A new preprocessor allows conditional compilation of these INFORMIX-ESQL/C statements: **\$include**, **\$define**, **\$undef**, **\$ifdef**, **\$ifndef**, **\$else**, **\$endif**.

## **Unlogged Temp Tables**

A WITH NO LOG option in the SELECT and CREATE TABLE statements lets you prevent logging of temporary tables in databases that use logging.

## **Multi-Statement PREPAREs**

A programmer can now PREPARE a string containing multiple SQL statements. This feature may improve performance by reducing the number of times the user program and database engine must exchange messages.

## New Run-Time Routines

The INFORMIX-ESQL/C library includes a number of new run-time routines that handle

- Numeric formatting (**rfmtlong**, **rfmtdouble**, **rfmtdec**)
- Message lookup (**rgetmsg**)
- Decimal rounding and truncation (**decround**, **dectrunc**)
- SQL database engine control (**sqlstart**, **sqlexit**, **sqlbreak**)

## Additional Enhancements to ESQL/C

In addition, Version 4.0 of INFORMIX-ESQL/C now supports

- Structures declared as INFORMIX-ESQL/C host objects
- Declaration of arrays as variables for types other than CHAR
- Host variable declaration with normal C initialization
- Standard C typedef expressions
- FREE syntax to release resources allocated to a PREPARE
- EXECUTE IMMEDIATE syntax to PREPARE, EXECUTE, and FREE a statement in one step

## Compatibility Issues

You should be aware of the following compatibility issues if you are upgrading to INFORMIX-ESQL/C Version 4.0 from an earlier version:

1. If you START an existing database as MODE ANSI, you must prefix objects with the names of their owners if they are referenced in existing queries that other users will execute. For example, the system catalogs must now have the prefix **informix**.
2. INFORMIX-ESQL/C no longer alters non-SQL source statements by converting uppercase letters in names of host variables to lowercase letters during processing. Thus, some compatibility problems may occur if existing programs assume the equivalence of uppercase and lowercase letters.
3. INFORMIX-ESQL/C no longer requires that indicator variables be short integers. Indicator variables can now be of any valid data type except datetime, interval, or locator. If you use non-short indicators in your programs, you should recompile and relink your programs to avoid errors.

4. The BEGIN WORK statement generates a run-time error unless it appears immediately after one of the following statements:  
CREATE DATABASE, DATABASE, START DATABASE, COMMIT WORK, ROLLBACK WORK.
5. In earlier releases of Informix products, synonym names applied only to the user who created them. This is no longer the case; synonym names now apply to all users, not just the user who created the synonym. This could affect existing applications that assume synonyms are local to the user, such as an application that uses a single synonym name to access a different view for each user or group of users.





# **INFORMIX-ESQL/C**

**Embedded SQL and Tools for C**

## **PROGRAMMER'S MANUAL**

**INFORMIX-ESQL/C**  
**Version 4.0**

**December 1989**  
**Part No. 200-626-0001-0**

THE INFORMIX SOFTWARE AND USER MANUAL ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE INFORMIX SOFTWARE AND USER MANUAL IS WITH YOU. SHOULD THE INFORMIX SOFTWARE AND USER MANUAL PROVE DEFECTIVE, YOU (AND NOT INFORMIX OR ANY AUTHORIZED REPRESENTATIVE OF INFORMIX) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION. IN NO EVENT WILL INFORMIX BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH INFORMIX SOFTWARE OR USER MANUAL, EVEN IF INFORMIX OR AN AUTHORIZED REPRESENTATIVE OF INFORMIX HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY. IN ADDITION, INFORMIX SHALL NOT BE LIABLE FOR ANY CLAIM ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH INFORMIX SOFTWARE OR USER MANUAL BASED UPON STRICT LIABILITY OR INFORMIX'S NEGLIGENCE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS, WHICH VARY FROM STATE TO STATE.

All rights reserved. No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems—without permission of the publisher.

Published by:  
Informix Software, Inc.  
4100 Bohannon Drive  
Menlo Park, CA 94025

**INFORMIX** and **C-ISAM** are registered trademarks of Informix Software, Inc.

UNIX is a trademark of AT&T.

IBM is a trademark of International Business Machines Corp.

## RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software Clause at 52.227-7013 (and any other applicable license provisions set forth in the Government contract).

Copyright © 1981-1990 by Informix Software, Inc.

# Table of Contents

## Preface

About this Manual.....	5
Chapter Summary.....	5
Syntax Conventions Used in this Manual.....	7
Preparing to Use INFORMIX-ESQL/C.....	9
Setting Environment Variables.....	9
The Demonstration Database.....	11
Creating the Demonstration Database.....	11
Restricted and Public Databases.....	12
Informix Databases.....	13
Informix Database Engines.....	13
Compatibility with Industry Standards.....	15

## Chapter 1. Using SQL

Chapter Overview.....	1-5
Relational Databases.....	1-6
SQL Identifiers.....	1-7
Owner Naming.....	1-8
Database Data Types.....	1-10
SQL Statement Overview.....	1-12
Data Definition and Administration.....	1-12
Data Manipulation.....	1-15
Cursor Management.....	1-16
Data Integrity.....	1-31
Dynamic Management.....	1-38
Locking.....	1-40
Row-Level Locking.....	1-41
Table-Level Locking.....	1-42
Wait for Locked Row.....	1-43
User Status and Privileges.....	1-44
Indexing Strategy.....	1-45
Auto-Indexing.....	1-48
Clustered Indexes.....	1-48
NULL Values.....	1-49
Default Values.....	1-49



The NULL in Expressions.....	1-50
The NULL in Boolean Expressions.....	1-51
The NULL in WHERE Clauses.....	1-51
The NULL in ORDER BY Clauses.....	1-52
The NULL in GROUP BY Clauses.....	1-53
The NULL Keyword in INSERT and UPDATE Statements.....	1-53
Views .....	1-54
Creating and Deleting Views.....	1-55
Querying through Views.....	1-55
Modifying through Views .....	1-56
Privileges with Views .....	1-57
Data Constraints Using Views .....	1-58
Outer Joins .....	1-59
Table Access by ROWID .....	1-60
TODAY and USER Functions.....	1-61

## Chapter 2. C Programming Using Embedded SQL

Chapter Overview.....	2-5
Header Files .....	2-6
Include Files .....	2-7
Host Variables .....	2-8
Indicator Variables .....	2-14
Embedding SQL Statements in C Routines.....	2-17
Error Handling and the <b>sqlca</b> Structure .....	2-18
Dynamic SQL Statements and the <b>sqllda</b> Structure .....	2-21
The <b>sqllda</b> Structure.....	2-21
Non-Parameterized SELECT Statements.....	2-23
Parameterized SELECT Statements.....	2-27
Parameterized Non-SELECT Statements.....	2-29
The ESQL/C Preprocessor .....	2-30
Preprocessor Support .....	2-30
Compiling ESQL/C Routines .....	2-32
Examples .....	2-35
demo1.ec .....	2-36
demo2.ec .....	2-37
demo3.ec .....	2-39
unload.ec.....	2-41

## Chapter 3. SQL Statements

SQL Statements.....	3-5
Statements Supported Only on INFORMIX-SE.....	3-6
Statements Supporting INFORMIX-OnLine Enhancements....	3-7
INFORMIX-ESQL/C Extensions to ANSI Syntax.....	3-7
Definition of Statements.....	3-11
The SELECT Statement.....	3-129
SELECT Clause.....	3-134
INTO Clause.....	3-137
FROM Clause.....	3-140
WHERE Clause.....	3-142
GROUP BY Clause.....	3-159
HAVING Clause.....	3-161
ORDER BY Clause.....	3-163
INTO TEMP Clause.....	3-165
UNION Operator.....	3-167
Functions in SQL Statements.....	3-170

## Chapter 4. SQL Data Types

Chapter Overview.....	4-5
Correspondence Between SQL and C.....	4-5
Data Conversion.....	4-7
CHAR Type.....	4-9
SMALLINT and INTEGER Types.....	4-10
SERIAL Type.....	4-10
SMALLFLOAT and FLOAT Types.....	4-11
DATE Type.....	4-11
MONEY Type.....	4-25
DECIMAL Type.....	4-25
DECIMAL Type Routines.....	4-27
Numeric Formatting Routines.....	4-50
Formatting Numeric Strings.....	4-50
DATETIME and INTERVAL Types.....	4-59
Declaration.....	4-60
DATETIME Columns.....	4-61
INTERVAL Columns.....	4-63
Fetching DATETIME and INTERVAL Values.....	4-65
Storing DATETIME and INTERVAL Values.....	4-66
DATETIME and INTERVAL Routines.....	4-67

## **Chapter 5. Library Functions**

Chapter Overview .....	5-5
Function Descriptions.....	5-5

## **Chapter 6. C Functions in ACE and PERFORM**

Chapter Overview .....	6-5
ACE Report Specification Files .....	6-6
Declaring C Functions .....	6-6
Calling C Functions .....	6-7
Compiling the Report Specification .....	6-9
PERFORM Form Specification Files .....	6-10
Calling the C Function .....	6-10
ON BEGINNING and ON ENDING .....	6-12
Compiling the Form Specification .....	6-13
Writing the C Program .....	6-14
C Program Structure .....	6-14
Input Parameters .....	6-17
Type Conversions.....	6-19
Return Values.....	6-19
Special PERFORM Library Functions .....	6-20
Compiling, Linking, and Running .....	6-31
Examples .....	6-32
ACE .....	6-33
PERFORM.....	6-37

## **Appendix A. Header Files**

## **Appendix B. System Catalogs**

## **Appendix C. Environment Variables**

## **Appendix D. Reserved Words**

## **Appendix E. The *stores* Database**

## **Appendix F. INFORMIX-ESQL/C Utility Programs**



**Appendix G. Outer Joins**

**Appendix H. Working with DATETIME and INTERVAL  
Data**

**Glossary**

**Error Messages**

**Index**





# Preface



## Preface Table of Contents

About this Manual.....	5
Summary of Chapters.....	6
Syntax Conventions Used in this Manual.....	7
Preparing to Use INFORMIX-ESQL/C .....	9
Setting Environment Variables .....	9
The Demonstration Database.....	11
Creating the Demonstration Database.....	11
Restricted and Public Databases.....	12
Informix Databases .....	13
Informix Database Engines .....	13
Compatibility with Industry Standards.....	15



# About this Manual

**INFORMIX-ESQL/C** (*Embedded SQL and Tools for C*) is designed for the C programmer who wants to create custom applications with database management capabilities, and for the advanced **INFORMIX-SQL** user who wants to use C functions in **PERFORM** forms and **ACE** reports. **INFORMIX-ESQL/C** contains preprocessors, C libraries, and header files that allow you to perform the following:

- Embed Structured Query Language (SQL) statements in C programs and routines
- Perform **DECIMAL**, **DATE**, **DATETIME**, and **INTERVAL** type conversions and manipulations
- Use C language utility functions
- Use C functions in **ACE** reports
- Use C functions in **PERFORM** forms

This manual assumes that the reader knows C programming and is familiar with the structure of relational databases. Chapter 6, “C Functions in **ACE** and **PERFORM**,” assumes that the reader is familiar with the **ACE** and **PERFORM** programs of **INFORMIX-SQL**, another member of the Informix Software, Inc. family of products. You can read about **ACE** and **PERFORM** in the ***INFORMIX-SQL User Guide***.

The underlying file and indexing structure of the database tables created through **INFORMIX-ESQL/C** or **INFORMIX-SQL** is built on **C-ISAM**. For more information about this indexed sequential access method for the C language, see the ***C-ISAM Programmer's Manual***.

---

**Note:** Two files supplement the information in the manual. **RELNOTES** describes performance differences from earlier versions of Informix products and how these differences may affect existing applications. **DOCNOTES** describes feature and performance topics not covered in the manual or modified since publication. Please examine these files as they contain vital information about application and performance issues. **RELNOTES** and **DOCNOTES** are located in the **\$INFORMIXDIR** release directory.

---



# Summary of Chapters

This manual includes the following chapters and appendixes:

- |                   |   |
|-------------------|---|
| <b>Preface</b>    | introduces <b>INFORMIX-ESQL/C</b> , discusses conventions used in syntax statements, and briefly describes the demonstration database.  |
| <b>Chapter 1</b>  | gives an overview of relational databases and how to use SQL. It provides information on Informix extensions to the American National Standards Institute (ANSI) standard SQL developed by IBM.   |
| <b>Chapter 2</b>  | describes how to write C programs containing embedded SQL statements. This chapter explains the structure of the C program and the relationships between the various SQL statements.  |
| <b>Chapter 3</b>  | describes, in alphabetical order, each of the SQL statements that you can use within a C program. Use this chapter as a reference for syntax and rules concerning the use of SQL statements.  |
| <b>Chapter 4</b>  | discusses the data types recognized by SQL and the relationships between them and C language data types. This chapter also contains detailed descriptions of library functions that permit manipulation of <b>DECIMAL</b> , <b>DATE</b> , <b>DATETIME</b> , and <b>INTERVAL</b> type variables. |
| <b>Chapter 5</b>  | describes additional C functions used for string manipulation.  |
| <b>Chapter 6</b>  | expands your power to customize <b>ACE</b> reports and <b>PERFORM</b> screen forms by describing how you can enter user-written C functions into expressions and control blocks. You can skip this chapter if you do not have <b>INFORMIX-SQL</b> .   |
| <b>Appendix A</b> | lists the header files used by <b>INFORMIX-ESQL/C</b> , <b>ACE</b> , and <b>PERFORM</b> .   |
| <b>Appendix B</b> | describes the system catalogs that form the data dictionary for a database.   |

- Appendix C** describes how to set the environment variables used by INFORMIX-ESQL/C.
- Appendix D** lists the reserved words that cannot be used as INFORMIX-ESQL/C program identifiers or database column names.
- Appendix E** describes the stores demonstration database used in this manual.
- Appendix F** describes the bcheck, dbload, dbschema, dbupdate, sqlconv, dbexport, and dbimport utilities provided with INFORMIX-ESQL/C.
- Appendix G** extends the description of outer joins begun in Chapter 1.
- Appendix H** provides additional information on working with DATETIME and INTERVAL data types.
- Glossary** defines common database terms used in this and other Informix products.
- Error Messages** contains an extensive listing of error codes and suggests remedies for correcting the errors.

## ***Syntax Conventions Used in this Manual***

The following notational conventions are used in syntax statements that appear in this manual.

- ABC** Any term in the syntax in uppercase letters is a keyword. Enter it exactly as shown, disregarding case. For example,  
  

**CREATE INDEX *indname***

means you must enter the keywords **CREATE INDEX** or **create index** without adding or deleting spaces or letters.
- abc*** Substitute a value for any term that appears in lowercase italic letters. In the previous example, you should substitute a value for *indname*. In each statement description, a section labeled "Explanation" describes what values you can substitute for italicized terms.

( ) Enter parentheses as shown. They are part of the syntax of a statement, not special symbols.

[ ] Do not enter brackets as part of an INFORMIX-ESQL/C statement; they surround any part of a statement that is optional. For example,

CREATE [UNIQUE] INDEX

indicates that you can enter either CREATE INDEX or CREATE UNIQUE INDEX.

| The vertical bar indicates a choice among several options. For example,

[ONE | TWO [THREE] | FOUR]

means that you can enter either ONE or TWO or FOUR and that, if you enter TWO, you also can enter THREE.

{ } When you must choose one of several options, the options are enclosed in braces and separated by vertical bars. For example,

{ONE | TWO | THREE}

means that you must enter ONE or TWO or THREE, and that you cannot enter more than one option.

ABC When one of several options is the default option, it appears underlined. For example,

[CHOCOLATE | VANILLA | STRAWBERRY]

means that you can select any of the three options, but that if you do not enter one, VANILLA is the default.

... Enter additional items like those preceding the ellipsis, if you choose. The ellipsis indicates that the immediately preceding item can be repeated indefinitely. For example,

*statement*

...

means that a series of statements can follow the one listed.



**Note:** The notation “(O)” sometimes follows the name of an SQL statement at the beginning of a syntax description in Chapter 3. This notation means that INFORMIX-ESQL/C supports additional options on the INFORMIX-OnLine database engine. Refer to the *INFORMIX-OnLine Programmer's Manual* for details of the additional functionality available with INFORMIX-OnLine.

## Preparing to Use INFORMIX-ESQL/C

This section explains the steps you must follow before running INFORMIX-ESQL/C and shows how to create the demonstration database. This section assumes that INFORMIX-ESQL/C is installed on your computer according to the installation instructions that come with the software.

As you follow the instructions in this manual, you will notice several different typefaces. The characters you type appear in a computer typeface that looks like this. Words for which you must supply values appear in italics *like this*.

## Setting Environment Variables

Before you can use INFORMIX-ESQL/C, you must set the following environment variables:

**INFORMIXDIR**      is the directory where INFORMIX-ESQL/C resides.

**PATH**              is the search path for programs you want to execute.

You can set these environment variables at the system prompt or in your **.profile** (Bourne shell) or **.login** (C shell) file.

- If you set these variables at the system prompt, you must reassign them the next time you log onto the system.

- If you set these variables in your **.profile** or **.login** file, they are assigned automatically every time you log onto the system.

Follow these steps to set your environment variables:

1. Log in.
2. Set the **INFORMIXDIR** environment variable as shown in the following examples. The instructions assume that **INFORMIX-ESQL/C** resides in the directory **/usr/informix**. If, for some reason, **INFORMIX-ESQL/C** resides in another directory, substitute that directory name for **/usr/informix**.

If you use the Bourne shell, enter:

```
INFORMIXDIR=/usr/informix
export INFORMIXDIR
```

If you use the C shell, enter:

```
setenv INFORMIXDIR /usr/informix
```

3. Include the directory containing **INFORMIX-ESQL/C** in your **PATH** environment variable.

If you use the Bourne shell, enter:

```
PATH=$INFORMIXDIR/bin:$P ATH
export PATH
```

If you use the C shell, enter:

```
setenv PATH ${INFORMIXDIR}/bin:${P ATH}
```

You can assign additional environment variables by following the instructions in Appendix C.

# The Demonstration Database

The INFORMIX-ESQL/C software includes a demonstration database named **stores**. Most of the examples in this manual are based on the **stores** database. This database, listed and described in detail in Appendix E, is concerned with the customers of, and the orders placed with, a wholesaler of sports equipment.

You can create the **stores** database in the current directory by typing **esqldemo**. This shell script removes any database labeled **stores** and installs the demonstration database and the example programs used in this manual.

The **stores** database consists of six tables:

<b>customer</b>	contains information about the retail stores that are the customers of the wholesaler.
<b>orders</b>	contains information about the orders placed.
<b>items</b>	contains information about the individual items in each order.
<b>stock</b>	contains a list of all stock items available from the wholesaler.
<b>manufact</b>	contains data on the manufacturer of the stock items.
<b>state</b>	contains a list of states.

## *Creating the Demonstration Database*

The following instructions tell you how to copy the demonstration database into a directory that you select.



1. Create a new directory for the demonstration database by entering

```
mkdir dirname
```

where *dirname* is the name of the new directory.

2. Make the new directory the current directory by entering

```
cd dirname
```

3. Create the demonstration database by entering

```
esqldemo
```

This command creates in the current directory the demonstration database and its sample programs, forms, and reports. If this command does not work, make sure you have set all your environment variables correctly.

You can run this command whenever you want to restore the original demonstration database after making changes.

## Restricted and Public Databases

In order for INFORMIX-ESQL/C to access a database, the current user must have READ and EXECUTE permissions for each directory in the database pathname. INFORMIX-ESQL/C provides database access privileges through the GRANT and REVOKE statements. However, these Informix database *privileges* have no effect on UNIX file *permissions*. (For more information, see the section "User Status and Privileges" in Chapter 1 and the discussions of GRANT and REVOKE in Chapter 3.)

- When you create a database, all users have READ and EXECUTE permissions for the database directory. You can restrict access to the database directory to yourself by typing

```
chmod 700 dirname
```

where *dirname* is the directory containing the database.

- You can make the database directory accessible to all users by typing

```
chmod 755 dirname
```

assuming that all directories above *dirname* have at least the same level of permissions.

---

**Caution!** It is not necessary to change permissions for the database directory or any other database files. Doing so may compromise or disable Informix access privileges.

---

## Informix Databases

The information that you can access with INFORMIX-ESQL/C must be stored in a *database*. A database is a collection of data, information about indexes, and system catalogs that describe the structure of the database.

Chapter 1 of this manual presents fundamental database concepts and describes how data is organized within the tables of an INFORMIX-ESQL/C database. Appendix B provides more information about INFORMIX-ESQL/C system catalogs.

## Informix Database Engines

You can use any Informix application development tool with either of two database engines: INFORMIX-SE or INFORMIX-OnLine.

INFORMIX-SE is based on C-ISAM, a library of C language calls that works in conjunction with UNIX to create and manipulate database files. INFORMIX-SE works automatically and transparently and it does not require any special instructions in your programs. Its easy setup and use make INFORMIX-SE an ideal engine for developing small- to medium-size applications that do not require maximum performance or an extensive range of data-integrity options.

INFORMIX-OnLine is a transaction-processing database engine that manages I/O operations directly so that performance is maximized. It is designed to handle the high performance requirements and integrity concerns of many large applications. Most applications can run on

either engine because few differences affect the programmer. However, **INFORMIX-OnLine** provides features that increase performance, extend available data types, and improve the administrative aspects of database management. The following features are available only with **INFORMIX-OnLine**:

- Raw I/O and optimized shared memory for greater performance
- High availability and automatic recovery
- Increased locking and process isolation options for greater integrity control
- Variable-length character data (VARCHARs)
- Binary Large Objects (BLOBs) that can be any type or amount of data
- Distributed query capability across multiple databases
- Distributed query capability across multiple **INFORMIX-OnLine** systems (optional)

**INFORMIX-OnLine** can substantially improve application performance by using direct I/O to raw storage devices and by tuning shared memory for maximum efficiency. **INFORMIX-OnLine** offers high data availability, which allows rapid recovery from system failures and provides you with logging choices, including the option of disk mirroring. **INFORMIX-OnLine** provides advanced concurrency control by allowing you to set the level of locking granularity and the level of user process isolation.

**INFORMIX-OnLine** supports three additional data types not available with **INFORMIX-SE**. VARCHARs are variable-length character columns (up to 255 bytes) that use disk space only as it is needed. The TEXT and BYTE data types are Binary Large Objects (BLOBs) capable of holding virtually any type or amount of data. TEXT can store ASCII text files with embedded control characters, such as documents generated through a word processor. BYTE can store any type of binary data, such as spreadsheets, program load modules, digitized images, or voice patterns.



**INFORMIX-OnLine** allows access to multiple databases on the same computer. You can write applications that retrieve data from multiple tables, even if the tables reside in different databases. You can use the combined data for general display purposes or as input to a customized report.

**INFORMIX-STAR** is an optional add-on product to **INFORMIX-OnLine** that allows you to access multiple **INFORMIX-OnLine** systems. With **INFORMIX-STAR** you can write queries that span multiple databases in different **INFORMIX-OnLine** systems across a network.

This manual provides you with information on how to use **INFORMIX-ESQL/C** with **INFORMIX-SE**. If you are planning to use **INFORMIX-OnLine**, you should refer to the *INFORMIX-OnLine Programmer's Manual* for information about programming issues. Notes are placed at appropriate locations in this manual to help clarify where **INFORMIX-OnLine** can provide additional functionality.

## ***Compatibility with Industry Standards***

The American National Standards Institute (ANSI) has established industry standards for Structured Query Language (SQL) and for embedded Structured Query Language (ESQL). **INFORMIX-ESQL/C** Version 4.0 conforms to the ANSI standard for ESQL (X3.168-1989), and to Level I of the ANSI Database Language SQL standard (X3.135-1986). It also conforms to Level II of the SQL standard, with two exceptions. This version of **INFORMIX-ESQL/C** does not support the following ANSI Level II features on **INFORMIX-SE**:

1. Module language
2. Serializable transactions

Support for ANSI standard Schema Language is available in **INFORMIX-SQL**. You can use the **CREATE SCHEMA AUTHORIZATION** statement with **INFORMIX-SQL** to specify an owner or grantor for a series of **CREATE** and/or **GRANT** statements. Refer to the *INFORMIX-SQL Reference Manual* for information on **CREATE SCHEMA**.





# **Chapter 1**

## **Using SQL**



# Chapter 1 Table of Contents

Chapter Overview .....	5
Relational Databases.....	6
SQL Identifiers .....	7
Owner Naming.....	8
Database Data Types.....	10
SQL Statement Overview.....	12
Data Definition and Administration.....	12
Data Manipulation.....	15
Cursor Management.....	16
SELECT Cursors.....	16
The SCROLL Cursor .....	23
The Cursor WITH HOLD .....	25
INSERT Cursors .....	28
Data Integrity .....	31
Transactions .....	32
Transaction Log File Maintenance.....	34
Audit Trails .....	35
Comparison of Transactions and Audit Trails.....	36
Locking Statements.....	37
Dynamic Management.....	38
Locking.....	40
Row-Level Locking.....	41
Row-Level Locking in Transactions .....	41
Table-Level Locking.....	42
Wait for Locked Row.....	43
User Status and Privileges.....	44
Indexing Strategy .....	45
Auto-Indexing .....	48
Clustered Indexes .....	48
NULL Values.....	49
Default Values.....	49
The NULL in Expressions .....	50
The NULL in Boolean Expressions .....	51
The NULL in WHERE Clauses .....	51

The NULL in ORDER BY Clauses .....	52
The NULL in GROUP BY Clauses .....	53
The NULL Keyword in INSERT and UPDATE Statements .....	53
Views .....	54
Creating and Deleting Views .....	55
Querying through Views .....	55
Modifying through Views .....	56
Privileges with Views .....	57
Data Constraints Using Views .....	58
Outer Joins .....	59
Table Access by ROWID .....	60
TODAY, CURRENT, and USER Functions .....	61



# Chapter Overview

Informix Software, Inc., has developed an extension to the Structured Query Language (SQL) developed by IBM. The additions that Informix has made to the language permit you to change databases, change the names of tables and columns, and increase the functionality of ANSI standard SQL statements.

You can check your SQL statements for ANSI compatibility by setting the `DBANSIWARN` environment variable. When `DBANSIWARN` is set, `INFORMIX-ESQL/C` writes a warning message to the screen at compile time whenever it encounters an Informix extension to ANSI standard syntax. At run time, `DBANSIWARN` causes `sqlwarn5` to be set to `W` when a non-ANSI statement is executed. See Appendix C, “Environment Variables,” for information on setting the `DBANSIWARN` environment variable.

Alternatively, you can use the `-ansi` flag to check for ANSI compatibility when you compile your `INFORMIX-ESQL/C` programs. See “Compiling ESQL/C Routines” in Chapter 2 for details of how to use the `-ansi` flag.

In the family of Informix database products, SQL plays several roles. In `INFORMIX-ESQL/C`, SQL is the database query language that you embed in C programs that allows you to access the database. In `INFORMIX-SQL`, SQL is both the interactive query language and the language you use to access the data for `ACE`, the `INFORMIX-SQL` report-writing program. You can read about these uses of SQL in the *INFORMIX-SQL User Guide*.

This chapter describes SQL and gives an overview of its statements. You can read about the full syntax and rules governing the SQL statements that you can use in `INFORMIX-ESQL/C` in Chapter 3. The statements are organized alphabetically.

The syntax displays use the conventions defined in the Preface. The statements described in Chapter 3 may differ from those described in Chapter 2 of the *INFORMIX-SQL Reference Manual*. The differences reflect the change in application between `INFORMIX-ESQL/C` and `INFORMIX-SQL`—that is, between the programming use and the interactive use.

# Relational Databases

SQL statements create and manipulate *relational databases*. A relational database consists of one or more *tables* that, in turn, are constructed of *rows* and *columns*. Each row contains a particular set of column values.

Databases are created as subdirectories of the current directory. The name of the subdirectory is the database name with the extension **.dbs**.

The database subdirectory contains 11 *system catalog* tables that define the database dictionary. It also contains the tables that constitute the database. Each of these tables is represented by data files and index files with the extensions **.dat** and **.idx**, respectively. The system catalogs are described in Appendix B.

You have the option of creating a MODE ANSI database that supports ANSI standards. When a database is created as MODE ANSI, transactions are initiated implicitly, and the referencing of objects by their owner names is enforced.

# SQL Identifiers

An SQL *identifier* is the name of an object and can consist of letters, numbers, and underscores (\_). The first character must be a letter. Unless qualified to the contrary, an identifier can have from 1 to 18 characters.

SQL identifiers are case-sensitive. Uppercase and lowercase letters in names of host variables are treated as distinct. Uppercase letters are not converted to lowercase during processing.

**Database** is a name that can have from 1 to 10 characters.

**Table** is a name that must be unique within the database.

**Column** is a name that must be unique within a table; there can be duplicate column names within a database. When column names within different tables are not unique, use the notation *table.column* to specify the intended column.

There can be no ambiguity between a C variable and an SQL identifier, since you must precede C variables that appear in SQL statements (see “Host Variables” in Chapter 2) with either a dollar sign (\$) or a colon (:). For example, if you define **lname** as a host variable and you want to refer to the database column of the same name, use **\$lname** or **:lname** for the host variable name:

---

```
$select lname into $lname from customer;
```

```
EXEC SQL select lname into :lname from customer;
```

---

**Note:** Use of EXEC SQL and the colon (:) before a host variable name conforms to ANSI standards.



## Owner Naming

In a database created as MODE ANSI, the name of each object (table, view, index, synonym, and constraint) is qualified by the name of the owner of the object. The combined *owner.object* must be unique within the database. The following rules apply to the naming of an object:

- *owner* is the login name of the owner of the object. The name must begin with a letter. It can contain underscores, letters, and numbers, and can be up to 8 characters long. Alternatively, the name can be a quoted string. This allows you to preserve uppercase characters in user names or to include a user name in which the first character is a digit. The quoted string can be up to 8 characters long.
- *object* is a valid identifier for a table, view, index, synonym, or constraint. The identifier must begin with a letter. It can contain underscores, letters, and numbers, and can be up to 18 characters long.

The format for naming an object in an SQL statement is as follows:

---

[*owner* . ]*object*

---

An object receives its owner when it is **CREATED**. You cannot change the ownership of an object.

By default, ownership is assigned to the individual who creates the object. However, a user with Database Administrator (DBA) privilege can create an object and assign ownership of the object to another user.

In a database created as MODE ANSI, you *must* specify *owner* when referring to an object created by another user. As with non-MODE ANSI databases, you *may* specify owner when referring to your own objects.

In the following example, the **UPDATE** statement modifies rows in the **stock** table owned by the user **james**:



---

```
$ update james.stock  
    set price = price * 1.05;
```

---

Quoted strings allow you to retain case sensitivity when case is important. In the following examples, the SELECT statements retrieve rows from different tables:

---

```
$ select * from "Smith".stock;  
  
$ select * from Smith.stock;
```

---

Note that "Smith" is in quotes in the first SELECT statement but not in the second. Because of the quotes, the case distinction is preserved in the first SELECT; therefore, the first SELECT retrieves rows from the **Smith.stock** table while the second SELECT retrieves rows from the **smith.stock** table.

The engine assumes an object is owned by the user if you do not include the *owner* prefix. As the owner of the system catalog tables is **informix**, you must include the owner name **informix** when querying each system catalog.

You do not have to supply owner names when working with a non-MODE ANSI database. However, if you specify the owner along with the object name, the engine will check for the accuracy of the owner name.

**Note:** In a MODE ANSI database, you receive an error if you do not use the *owner.object* naming convention to refer to an object owned by another user. If you start a database as MODE ANSI, you must modify existing queries that reference a table, view, or synonym owned by another user to include the *owner* prefix.

# Database Data Types

You must assign a data type to every column in the database. See the description of the CREATE TABLE statement in Chapter 3 for details.

Valid data types in SQL are as follows:

**CHAR(*n*)** is a character string of length *n* (where  $1 \leq n \leq 32,511$ ). If you do not specify *n*, CHAR(1) is assumed.

**CHARACTER(*n*)** is a synonym for CHAR.

**SMALLINT** is a whole number from -32,767 to +32,767.

**INTEGER** is a whole number from -2,147,483,647 to +2,147,483,647.

**INT** is a synonym for INTEGER.

**DECIMAL(*m*,*n*)** is a decimal floating point number with *m* ( $\leq 32$ ) significant digits (the precision) and *n* ( $\leq m$ ) digits to the right of the decimal point (the scale). When you give values for both *m* and *n*, the decimal variable has fixed-point arithmetic. All numbers less than  $0.5 \times 10^{-n}$  in absolute value have the value zero. The largest absolute value of a variable of this type that you can store without an error is  $10^{m-n} - 10^{-n}$ .

The second parameter is optional and, if missing, the variable is treated as a floating decimal. This means that DECIMAL(*m*) variables have a precision of *m* and a range in absolute value from  $10^{-128}$  to  $10^{126}$ . If no parameters are designated, DECIMAL is treated as DECIMAL(16), a floating decimal.

**DEC(*m*,*n*)** is a synonym for DECIMAL.

**NUMERIC(*m*,*n*)** is a synonym for DECIMAL.

**SMALLFLOAT** is a binary floating-point number corresponding to the float C data type. The range of values for a SMALLFLOAT data

	type is the same as the range of values for the C float data type on your machine.
REAL	is a synonym for SMALLFLOAT.
FLOAT[(n)]	is a binary floating-point number corresponding to the double C data type. The range of values for a FLOAT data type is the same as the range of values for the C double data type on your machine. You can use <i>n</i> to specify the precision of a FLOAT data type, although INFORMIX-ESQL/C ignores the precision. <i>n</i> must be a whole number between 1 and 14.
DOUBLE PRECISION	is a synonym for FLOAT.
MONEY[(m[,n])]	can take two parameters like the DECIMAL data type. The limitation on values for columns of type MONEY( <i>m</i> , <i>n</i> ) is the same as for columns of type DECIMAL( <i>m</i> , <i>n</i> ). The type MONEY( <i>m</i> ) is defined as DECIMAL( <i>m</i> , 2) and, if no parameter is given, MONEY is taken to be DECIMAL(16, 2). Regardless of the number of parameters, the data type MONEY will always be treated as a fixed decimal number.
SERIAL[(n)]	is a sequential integer assigned automatically by SQL. You can assign an initial value <i>n</i> . The default starting integer is 1.
DATE	is a date entered as a character string in one of the formats described in Chapter 4 and stored as an integer number of days since December 31, 1899.
DATETIME <i>first TO last</i>	stores a moment in time with the precision <i>first to last</i> . A DATETIME column consists of a contiguous sequence of the following fields: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and FRACTION( <i>n</i> ) of a second. DATETIME is described in greater detail in Chapter 4 and Appendix H.



**INTERVAL** *first TO last* stores a span of time with the precision *first to last*. An **INTERVAL** column consists of a contiguous sequence of one of the following two lists of fields: either **YEAR** and **MONTH**, or **DAY**, **HOURL**, **MINUTE**, **SECOND**, and **FRACTION(*n*)** of a second. **INTERVAL** is described in greater detail in Chapter 4 and Appendix H.

---

**INFORMIX-OnLine** supports additional data types. Refer to the *INFORMIX-OnLine Programmer's Manual* for more information.

---

## SQL Statement Overview

Most of the SQL statements used with **INFORMIX-ESQL/C** fall into the following categories:

- Data definition and administration
- Data manipulation
- Cursor management
- Data integrity
- Dynamic management

### *Data Definition and Administration*

Data definition and administration statements include those that create, drop, and control access to a database and its tables, views, synonyms, and indexes, and modify or rename tables and columns. Of this list, only the **DATABASE** statement is required before manipulating the data of an existing database.

#### **CREATE DATABASE**

creates a database directory, sets up the system catalogs, and makes the new database the *current database*. There can be no more than one current database at any time.



**DATABASE**

selects a database and makes it the current database. There can be no more than one current database at any time. The **DATABASE** statement has the option **EXCLUSIVE** that opens the database in an **EXCLUSIVE MODE** and allows only the Database Administrator (DBA) to have access to the database. This option is particularly useful when checking or archiving the database.

**CLOSE  
DATABASE**

closes the current database files and leaves no database current. The only SQL statements permitted when there is no current database are those in the following list:

- **CREATE DATABASE**
- **DATABASE**
- **DROP DATABASE**
- **START DATABASE**

**DROP  
DATABASE**

deletes all tables, indexes, system catalogs, and the database subdirectory. If no other files are present in the database directory, the subdirectory is also deleted.

**CREATE TABLE**

creates a table and defines the columns and their data types.

**ALTER TABLE**

adds and drops columns from a table and modifies data types of existing columns.

**RENAME TABLE**

changes the name of a table.

**DROP TABLE**

deletes all data and indexes for a table and erases its entry in the system catalogs.

**CREATE VIEW**

defines a table selected from rows and columns of existing tables and views. As the underlying tables change, so does the view built upon them. See the section "Views" later in this chapter for an explanation of how to use views.

**DROP VIEW** deletes the definition of the view from the system catalogs along with any views defined in terms of the one that is dropped. The underlying tables are unaffected.

**CREATE SYNONYM** defines an alternative name for a table or a view. For INFORMIX-ESQL/C programs, the creator is the user who runs the program that creates the synonym.

**DROP SYNONYM** deletes a synonym from the system catalogs.

**RENAME COLUMN** changes the name of a column.

**CREATE INDEX** creates an index on one or more columns of a table.

**DROP INDEX** deletes a previously CREATED index.

**ALTER INDEX** allows you to modify a table so that the rows are in the same order as the index. The index is called a *clustered* index.

See the section "Indexing Strategy" later in this chapter for more information on indexes.

A user has access to the database and to specific columns within a table only when the DBA or the owner of the table specifically grants these privileges. SQL provides the following statements affecting data access:

**GRANT** grants database access privileges to specific users or to the public.

**REVOKE** removes database access privileges from specific users or from the public. See the section "User Status and Privileges" later in this chapter for further information.

## **UPDATE STATISTICS**

updates the system catalogs by determining and inserting the number of rows in the indicated tables. **INFORMIX-ESQL/C** uses this information in optimizing queries, but does not automatically update the system catalogs after each **INSERT** or **DELETE**.

## ***Data Manipulation***

The most frequently used statements are the data manipulation statements that follow:

- DELETE**        deletes one or more rows from a table.
- INSERT**        adds one or more rows to a table.
- SELECT**        retrieves data from one or more tables.
- UPDATE**        modifies the data in one or more rows of a table.

**SELECT** is the most important and the most complex SQL statement. The following examples illustrate its use, and the syntax is defined in detail in Chapter 3.

---

```
$ select lname, company
      into $p_lname, $p_company
      from customer
      where customer_num = 101;
```

---

This statement queries the **customer** table and returns the single row for which the customer number is 101. From that row, it selects and places the values in the columns corresponding to the contact's last name and company name in the host variables **p\_lname** and **p\_company**.



---

```
$  select quantity, total_price
    into $quantity, $total_price
    from items
    where order_num = 1001;
```

---

This statement shows another **SELECT** statement that returns a single row. In this example, the host variables **quantity** and **total\_price** have the same identifier as the corresponding columns in the **items** table. There is no conflict here since the prefixed **\$** distinguishes the column name from the host variable. A **SELECT** statement that returns a single row is called a *singleton* **SELECT** statement and can stand alone. See the following section, “Cursor Management,” for how to handle **SELECT** statements that return more than one row.

## Cursor Management

**INFORMIX-ESQL/C** provides two basic types of cursors:

- A **SELECT** cursor, which you must use to handle a **SELECT** statement that returns more than one row.
- An **INSERT** cursor, which you can use to insert rows into the database as a block.

The section “**SELECT** Cursors” describes how to associate a cursor with a **SELECT** statement and gives examples of the uses of the **SELECT** cursor. The section “**INSERT** Cursors” explains how to associate a cursor with an **INSERT** statement and describes the advantages of using an **INSERT** cursor.

### SELECT Cursors

In the preceding examples, the **SELECT** statement returned exactly one row, and the values returned to the host variables were unambiguous. When more than one row can be returned, you need to have a device to distinguish one row from another. This device is called a *cursor*.

The set of rows returned by a **SELECT** statement is called the *active set* for the statement. Within an **INFORMIX-ESQL/C** program, you can work with only one row of the active set at a time. This row is called the *current row* and is referenced by a cursor.



A cursor can be in one of two states: open or closed. When it is in an open state, the cursor points to the current row, between two rows, before the first row, or after the last row. When it is in a closed state, the cursor no longer is associated with an active set, although it remains associated with the **SELECT** statement.

The following sections describe how to use the cursor management statements to process rows returned by a **SELECT** statement. For complete information on the syntax of each statement, see Chapter 3.

***Associating a Cursor with a SELECT Statement.*** You use the **DECLARE** statement to name a cursor and associate it with a **SELECT** statement. In the **DECLARE** statement, you specify the type of cursor you want to use:

- A regular cursor allows rows to be retrieved from the active set in consecutive order. In addition, you must declare a regular cursor **FOR UPDATE** when you plan to delete or update the current row in the active set using **WHERE CURRENT OF cursor-name**. If the database has transactions, a **COMMIT WORK** or **ROLLBACK WORK** closes a regular cursor.
- A **SCROLL** cursor allows rows to be retrieved from the active set in random order.
- Unlike a regular or **SCROLL** cursor, a regular or **SCROLL** cursor declared as **WITH HOLD** is not closed at the end of a transaction.

For example, the following **DECLARE** statement associates a **SCROLL** cursor named **q\_curs** with a **SELECT** statement that retrieves all the rows from the **customer** table:

---

```
$ declare q_curs scroll cursor for
      select * from customer;
```

---

The following **DECLARE** statement associates a regular cursor with a **SELECT** statement that retrieves customer rows based on a last name that the user supplies:

---

```
$  declare cust_curs cursor for
    select * from customer
    where lname matches $last_name;
```

---

A cursor name has meaning only from the point at which it is **DECLARED** to the end of the source code file. This means that the **DECLARE** statement for a cursor must physically or logically appear before any statement that references it.

**Retrieving and Processing Rows.** Once you have **DECLARED** a cursor for a **SELECT** statement, you can use the **OPEN**, **FETCH**, and **CLOSE** statements to retrieve and process the rows specified by the **SELECT** statement. Use these statements when you need to explicitly control the behavior of a cursor:

**OPEN** puts the cursor in an open state with regard to the **SELECT** statement. The **OPEN** statement sets the conditions of the **SELECT** statement based on the current values of the host variables, and it may initiate processing needed to execute the **SELECT** statement, such as creating a temporary table to handle an **ORDER BY** clause. While the cursor is in an open state, subsequent changes to any host variables do not affect the active set of rows that will be returned by the **SELECT** statement.

**FETCH** populates a program data buffer (if the required row is not already there), moves the cursor to the specified row (as requested by **FIRST**, **LAST**, **NEXT**, **PRIOR** or **PREVIOUS**, **ABSOLUTE *n***, or **RELATIVE *m***), and retrieves the values from that row. If a **FETCH** statement moves the cursor before the first row or after the last row, the **sqlcode** component of the **sqlca** record has the value **SQLNOTFOUND** (= 100). (See "Error Handling and the **sqlca** Structure" in Chapter 2 for more information.) **SQLNOTFOUND** indicates that either end of the active list has been reached.

**CLOSE** puts the cursor in a closed state and releases the active set. No statements referring to the cursor, other than **OPEN**, are operative.

To illustrate these statements, consider the following DECLARE statement:

---

```
$ declare x cursor for
    select order_num, order_date
      from orders
     where paid_date is null
        and ship_date > $p_date
    for update of paid_date;
```

---

This statement names a cursor **x** and associates it with the **SELECT** statement following the **FOR** keyword. The **SELECT** statement returns the order number and order date for those unpaid orders whose shipping date was later than the date in the host variable **p\_date**. The statement also enables a future **UPDATE** statement to modify the **paid\_date** column. The **DECLARE** statement does not run the query; it simply assigns the cursor to the **SELECT** statement:

---

```
$ open x;
```

---

When you execute the **OPEN** statement later in your program, you set the **WHERE** clause of the **SELECT** statement based on the value of the host variable **p\_date** at the time of the **OPEN** statement.

---

```
$ fetch x into $order_num, $order_date;
```

---

The **FETCH** statement populates the program data buffer, moves **x** to point to the first row, and fills the host variables **order\_num** and **order\_date** with the values of the columns **order\_num** and **order\_date** in the first row:



---

```
$ update orders
    set paid_date = TODAY
    where current of x;
```

---

The UPDATE statement changes the value of the date the order was paid to today's date in the current (first) row. The cursor remains pointing to the first row:

---

```
$ close x;
```

---

The cursor **x** is now put into a closed state, and the active set is effectively dissolved. An immediate FETCH statement would be an error, because a cursor in a closed state cannot point to anything. If, at a later time, you should execute the statement

---

```
$ open x;
```

---

the cursor **x** would be put back into an open state with a new active set that depends on the value of the host variable **p\_date** when the OPEN was executed.

***Deleting or Updating the Current Row.*** You can use special forms of DECLARE, DELETE, and UPDATE to delete or update the current row in an active set. You must use a non-SCROLLing cursor to process the rows returned by a SELECT statement, and the SELECT statement cannot include an ORDER BY clause. To delete a row in an active set, you must include a FOR UPDATE clause in the DECLARE statement for a non-SCROLLing cursor and a WHERE CURRENT OF clause in a subsequent DELETE statement. See the following example.



---

```

/* Prompt user then read name from terminal. */
printf("Enter a last name: ");
getline(last_name, 20);

$ declare q_curs cursor for
    select * from customer
    where lname matches $last_name
    for update;

$ open q_curs;
for (;;)
{
$   fetch q_curs into $cust_rec;
    if (sqlca.sqlcode != 0)
        break;

    /* Display customer values here. */
    ...

    printf("Do you want to delete this customer
        (y/n) ? ");
    getline(answer, 1);

    if (answer[0] == 'y')
$       delete from customer where current of q_curs;

    }

    ...

$   close q_curs;

```

---

The cursor remains between rows after a **DELETE WHERE CURRENT OF** statement is executed. This means that you cannot refer to the cursor in another **DELETE** or **UPDATE** statement until you use a **FETCH** statement to advance the cursor to the next row.

You can update the current row if you include a FOR UPDATE clause in the DECLARE statement for a non-SCROLLing cursor and a WHERE CURRENT OF clause in a subsequent UPDATE statement. The following example allows the user to update the address information in the current row:

---

```
$ declare q_curs cursor for
    select * from customer
    for update;

$ open q_curs;
for(;;)
{
$   fetch q_curs into $cust_rec;
   if (sqlca.sqlcode != 0)
       break;

   /* Display customer values here. */
   ...

   printf("Do you want to change the customer's
        address (y/n) ? ");
   getline(answer, 1);

   if (answer[0] == 'y')
       {

/* Input the new values here. */
...

        update customer
        set address1 = $cust_rec.address1,
            address2 = $cust_rec.address2,
            city = $cust_rec.city,
            state = $cust_rec.state,
            zipcode = $cust_rec.zipcode
        where current of q_curs;
        }
        ...
    }
$ close q_curs;
```

---

If you specify one or more columns in the FOR UPDATE clause of the DECLARE statement, you can only update those columns in a subsequent UPDATE WHERE CURRENT OF statement. If you do not list columns in a FOR UPDATE OF *column-list* clause, you can update any column retrieved in the select.

The following example allows the user to update the **fname** and **lname** columns of the current row:

---

```
$ declare q_curs cursor for
      select * from customer
      for update of fname, lname;

$ open q_curs;
  for(;;)
  {
$      fetch q_curs into $cust_rec;
      if (sqlca.sqlcode != 0)
          break;

      /* Display customer values here. */
      ...

      printf("Do you want to change the contact's
              name (y/n) ? ");
      getline(answer, 1);

      if (answer[0] == 'y')
      {
          /* Input the new customer values here. */
          ...

$          update customer
              set fname = $cust_rec.fname,
                lname = $cust_rec.lname
              where current of q_curs;
      }
  }
$ close q_curs;
```

---

The position of the cursor does not change after an **UPDATE WHERE CURRENT OF** statement is executed.

## The SCROLL Cursor

When you need to process the rows returned by a **SELECT** statement in random order, you must **DECLARE** a **SCROLL** cursor.

When you initially fetch a row with a **SCROLL** cursor, all the rows in the active set up to and including the fetched row are placed in a temporary file and remain there until you close the cursor. If you then fetch the same row or any row prior to it, **INFORMIX-ESQL/C** retrieves the row from the temporary file instead of from the database.

The temporary file allows you to retrieve rows in a random order. However, it also means that subsequent changes to the database may not be reflected in the active set used by a SCROLL cursor. Thus, you cannot declare a SCROLL cursor FOR UPDATE. Instead, you must declare a regular cursor (or a cursor WITH HOLD) FOR UPDATE when you plan to perform a subsequent UPDATE WHERE CURRENT OF or DELETE WHERE CURRENT OF action.

The following example DECLAREs a SCROLL cursor and fetches the last row in the active set.

---

```
$  declare q_curs scroll cursor for
      select order_date from orders
      where customer_num = 118;

...

$  open q_curs;

...

$  fetch last q_curs into $order_date;
```

---

All FETCH statements except the default FETCH NEXT statement require that you first DECLARE a SCROLL cursor. The default FETCH statement works with all cursors.

You cannot DECLARE a SCROLLing cursor FOR UPDATE. This means you cannot use the WHERE CURRENT OF keywords to UPDATE or DELETE a row fetched by a SCROLL cursor.

**Note:** When you open a cursor that identifies a SELECT statement containing a host variable, INFORMIX-ESQL/C runs the SELECT statement with the current value of the host variable. In the following example, the active sets produced by the first and second OPEN statements differ because the value of last\_name changes from Baxter to Grant:



---

```
        strcpy(last_name, "Baxter");

$   declare q_curs scroll cursor for
        select * from customer
        where lname matches $last_name;

...

$   open q_curs;

...

$   close q_curs;

        strcpy(last_name, "Grant");

$   open q_curs;
```

---

***SCROLL Cursors and a MODE ANSI Database.*** The SCROLL cursor is an Informix extension to ANSI standard syntax. You can use a SCROLL cursor with a database created as MODE ANSI. However, the use of the SCROLL keyword will cause a warning message if the DBANSIWARN environment variable is defined or the program is compiled with the **-ansi** flag.

## The Cursor WITH HOLD

In a database with transactions, the COMMIT WORK and ROLLBACK WORK operations end a transaction and release all row and table locks. In addition, these statements close all cursors except the cursor WITH HOLD.

Unlike other cursors, you can OPEN a cursor WITH HOLD outside a transaction, and you must explicitly CLOSE the cursor. While this characteristic is useful for certain types of applications, you should be careful when you include a cursor WITH HOLD in your program.

The following example outlines a typical program structure that uses a cursor WITH HOLD. A similar program in a MODE ANSI environment would not include the BEGIN WORK statement.

---

```

$ declare c_orders cursor with hold for
    select * from customer;
$ open c_orders;
$ for(;;)
{
$     fetch c_orders into $cust_rec;
    if (sqlca.sqlcode != 0)
        break;
$     begin work;
$     declare c_items cursor for
        select * from orders
        where customer_num = $cust_rec.customer_num
    for update;
$     open c_items;
    for(;;)
    {
$         fetch c_items into $order_rec;
        if (sqlca.sqlcode != 0)
            break;

        /* Update the order row here. */
        ...

$         update orders set * = $order_rec
        where current of c_items;
    }
$     close c_items;
$     commit work;
    }
$ close c_orders;

```

---

In this program, the cursor WITH HOLD provides the following advantages:

- You can open the **c\_orders** cursor outside a transaction. The **BEGIN WORK** statement appears after you **OPEN** the **c\_orders** cursor and perform a **FETCH**, and before you **OPEN** the **c\_items** cursor.
- The **COMMIT WORK** at the end of each iteration of the main loop does not close the **c\_orders** cursor. The cursor remains open to **FETCH** the next master row after the **COMMIT WORK** has closed the **c\_items** cursor and released all locks. The updated rows are now available to other users on the system.

If you do not use a cursor WITH HOLD, you must place the **BEGIN WORK** and **COMMIT WORK** statements completely outside the main loop. You would open both the **c\_orders** and **c\_items** cursors, **FETCH**

all master rows, and FETCH and UPDATE all detail rows within the single transaction. This approach has the following drawback: it holds all locks for the duration of the entire main loop. If your program updates a large number of rows, you can exceed the limits your operating system places on the number of rows that can be locked at one time. In addition, the locked rows are unavailable to other users on the system.

***The Cursor WITH HOLD and Locks.*** As noted previously, in a database with transactions, you must open a cursor not declared as WITH HOLD that is FOR UPDATE inside a transaction. Thus, any UPDATE or DELETE actions that are based on a cursor that is not declared as WITH HOLD occur within a transaction. You can always roll back the actions if necessary.

You cannot roll back an UPDATE or DELETE operation performed with a cursor WITH HOLD outside a transaction, because an SQL operation that takes place outside a transaction is treated as a singleton transaction. That is, the UPDATE or DELETE operation is preceded by an implicit BEGIN WORK and followed by an implicit COMMIT WORK. Any locks acquired during a singleton transaction are released as soon as the operation ends. Outside a transaction, no locks are retained from statement to statement.

In short, the cursor WITH HOLD is designed to provide a natural way of doing a read-only, forward scan over a table, independent of transaction boundaries. Other uses are permitted, but the programmer must understand the implications and risks of circumventing the transaction mechanism.

---

**Note:** See the *INFORMIX-OnLine Programmer's Manual* for a discussion of cursors WITH HOLD and locks on the INFORMIX-OnLine database engine.

---

***The Cursor WITH HOLD and a MODE ANSI Database.*** The cursor WITH HOLD is an Informix extension to ANSI standard syntax. You can use a cursor WITH HOLD with a database created as MODE ANSI. However, the use of the WITH HOLD keywords will cause a warning message if the DBANSIWARN environment variable is defined or the program is compiled with the -ansi flag.



## INSERT Cursors

You can associate a cursor with an INSERT statement as well as with a SELECT statement. The INSERT cursor permits data to be more efficiently inserted into a database by buffering the data in memory and writing to the disk only when the buffer is full. An INSERT cursor must occur within a transaction in a database that has transactions.

The following statements allow you to declare and manipulate an INSERT cursor. (For complete information about the syntax of each statement, see Chapter 3.)

- DECLARE** associates a cursor with an INSERT statement. (The INSERT statement may not contain an embedded SELECT statement.) You cannot DECLARE a SCROLL INSERT cursor or an INSERT cursor WITH HOLD.
- OPEN** sets up an insert buffer for an INSERT cursor.
- PUT** stores a row in the INSERT buffer for later insertion into the database. When you fill the buffer (by executing a series of PUT statements), **INFORMIX-ESQL/C** automatically inserts the rows into the appropriate table as a block.
- FLUSH** forces **INFORMIX-ESQL/C** to insert the buffered rows into the database without closing the INSERT cursor. You can force the insertion using the FLUSH statement, but you cannot delay insertion by not using the FLUSH statement.
- CLOSE** flushes the insert buffer and closes the INSERT cursor.

For databases with transactions, you must issue the OPEN, PUT, FLUSH, and CLOSE statements within a transaction.



For example, you can use these cursor management statements to insert customers into the **customer** table block by block.

---

```
$ declare ins_curs cursor for
    insert into customer values ($cust_rec);

$ open ins_curs;

do
{
    /* Input the new customer values here. */
    ...

    cust_rec.customer_num = 0;

$ put ins_curs;

    printf("Do you want to enter another");
    printf(" customer (y/n) ? ");
    getline(answer, 1);
} while (answer[0] == 'y');

$ close ins_curs;
```

---

This example includes the following statements:

- The **DECLARE** statement associates a cursor called **ins\_curs** with an **INSERT** statement that inserts a row into the **customer** table.
- The **OPEN** statement sets up the insert buffer for the **INSERT** cursor.
- The do-while loop includes statements to accept user input (not shown) and inserts that information into the **customer** table, block by block. Specifically, the data entry code allows the user to enter customer information and stores the information in the **cust\_rec** record. The **PUT** statement stores the current values in the **cust\_rec** record in the insert buffer. If the insert buffer becomes full as the result of a **PUT** statement, **INFORMIX-ESQL/C** automatically inserts the rows into the **customer** table as a block.
- The **CLOSE** statement inserts any rows that remain in the insert buffer into the **customer** table and closes the **INSERT** cursor.

When you use an insert cursor, you should **CLOSE** the cursor to insert any buffered rows into the database before allowing your program to end. The user may lose data if the cursor is not closed properly.

---

```
$ declare ins_curs cursor for
    insert into customer values ($cust_rec);

$ open ins_curs;

do
{
    /* Input the new values here. Set exit_flag if the
       user wants to exit. */
    ...

    if (exit_flag)
    {
        $ close ins_curs;
        exit(0);
    } while (answer[0] == 'y');

    cust_rec.customer_num = 0;

$ put ins_curs;

    printf("Do you want to enter another");
    printf(" customer (y/n) ? ");
    getline(answer, 1);
}

$ close ins_curs;

...
```

---

You can determine whether **INFORMIX-ESQL/C** successfully executes a **PUT**, **FLUSH**, or **CLOSE** statement by examining the values of the **sqlca.sqlerrd[2]** variable. (See "Error Handling and the **sqlca** Structure" in Chapter 2 for more information on these variables.)

- If **INFORMIX-ESQL/C** simply puts a row in the insert buffer, it assigns the following values to these global variables:

```
sqlca.sqlcode = 0
sqlca.sqlerrd[2] = 0
```

- If **INFORMIX-ESQL/C** successfully inserts a block of rows into the database as a result of a **PUT**, **FLUSH**, or **CLOSE** statement, it assigns the following values:

```
sqlca.sqlcode = 0
sqlca.sqlerrd[2] = the number of rows inserted
```

- If, as a result of a **PUT**, **FLUSH**, or **CLOSE** statement, **INFORMIX-ESQL/C** is unsuccessful in its attempt to insert an entire block of rows into the database, it assigns the following values:

```
sqlca.sqlcode = a negative number corresponding
                to the error message
sqlca.sqlerrd[2] = the number of rows successfully inserted
```

## ***Data Integrity***

SQL provides data integrity in several ways. You can prevent users of the database from changing the same row at the same time. You do this by explicitly locking a table or by accessing the row from within a transaction. You also can guarantee data integrity by using the recovery feature, which is implemented with transactions, or by using audit trails.

SQL provides data integrity by implementing the idea of transactions. A transaction is a series of operations on a database (SQL statements) that you want to be completed entirely or not at all. Examples of transactions are abundant in bookkeeping where several operations on several different accounts must be made as a unit or the books will be out of balance. Your **INFORMIX-ESQL/C** programs can ensure the data integrity of your database by using the following statements:

<b>BEGIN WORK</b>	marks the beginning of a transaction in a database that is not created as <b>MODE ANSI</b> .
<b>COMMIT WORK</b>	marks the end of a transaction by authorizing all changes to the database since the beginning of the transaction.
<b>ROLLBACK WORK</b>	marks the end of a transaction by revoking all changes to the database since the beginning of the transaction.



<b>START DATABASE</b>	initiates a new transaction log file.
<b>ROLLFORWARD DATABASE</b>	uses a transaction log file to restore a database from backup.

## Transactions

A transaction is a series of operations on a database (SQL statements) that you want to be completed entirely or not at all. SQL provides several statements that give you the ability to protect the integrity of your database by treating a transaction as a unit.

The first statement is the **CREATE DATABASE** statement including the **WITH LOG IN** clause. This statement creates not only the database but also a transaction log file that keeps track of all modifications to the database.

When a database uses logging, the default for temporary tables is to log them as well. You can prevent logging of temporary tables by including the **WITH NO LOG** keywords in your **CREATE TEMP TABLE** and **SELECT INTO TEMP** statements. See Chapter 3 for details.

When you want to perform a series of operations that you consider a unit, you issue the **BEGIN WORK** statement. This statement causes all subsequently altered rows of the database tables to be locked against modification by others (although others may view them). When you are satisfied that the series of operations has produced the desired results, you terminate the transaction with a **COMMIT WORK** statement.

If you are not happy with the results, you can terminate the transaction with a **ROLLBACK WORK** statement. This statement restores the database to the state that existed when you issued the **BEGIN WORK** statement—with an important exception. All database definition statements that alter the number or names of tables, the number or names of columns, the data types, or the indexes are treated as singleton transactions; that is, if they were executed successfully, they are committed and are not rolled back. Both the **COMMIT WORK** and the **ROLLBACK WORK** statements unlock the rows and make them accessible for modification by others.



The number of rows that can be locked at one time by all users is limited. The actual limit depends on your operating system. Try to restrict the definition of a transaction to a few statements that involve only a few rows. If you expect that the number of rows entering into the transaction will be large, LOCK the tables involved until the transaction is completed.

**Note:** *The BEGIN WORK, COMMIT WORK, and ROLLBACK WORK statements cannot work unless you CREATED the database with the WITH LOG IN option or STARTed the database with a transaction log file. A database created without a transaction log is described as a database "without transactions." A database created with a transaction log is described as a database "with transactions."*

If you open or create a database with transactions, but do not execute the BEGIN WORK statement, INFORMIX-ESQL/C treats each statement as a singleton transaction. Each statement, if it executes successfully, is committed, and the database is permanently altered. If the statement fails, there is an automatic rollback to the status before the statement.

You must execute cursor manipulation statements inside a transaction if your database was created or started with transactions. You should execute the BEGIN WORK statement before opening a cursor. The COMMIT WORK and ROLLBACK WORK statements close all open cursors.

You cannot ROLLBACK any of the Data Definition statements nor the GRANT or REVOKE statements. *Do not use these statements within a transaction.*

If you create a database without transactions, there is no automatic recovery from a situation where you want to treat several database operations as a single unit of work. For example, under transactions, you can UPDATE several rows as a single unit of work. If the UPDATE fails after changing some rows but not all, you can ROLLBACK the transaction to the original state where no rows are modified. Without transactions, you must take explicit action to restore the updated rows.

If you want a database to have implicit transactions, you must create (or start) the database as MODE ANSI. A MODE ANSI database must have a transaction log, and all SQL statements are automatically within a transaction. Singleton transactions do not exist under implicit transactions, and you should not use the BEGIN WORK statement.

You explicitly terminate the transaction when you issue a COMMIT WORK or ROLLBACK WORK statement, and initiate a new transaction with the next statement.

## Transaction Log File Maintenance

The transaction log file can become quite large and, periodically, the Database Administrator may want to archive it on tape and initiate another log file. At the same time, the DBA also should create a backup of your database. Generally speaking, every log file must have a corresponding archive copy of the database. After backing up the log file and the database, the DBA must specify an empty log file. In order to reuse the same log file, the DBA can create an empty log file with the same name as the old one. You can do this with the following command:

```
cat /dev/null > logfile
```

In order to change the name of the log file, the DBA must execute the START DATABASE statement just before making a backup of the database. The START DATABASE statement locks the database in EXCLUSIVE MODE while it is operating so that no further changes can be made. If START DATABASE fails, no database is open.

If the database is without transactions and you want to use transactions, the DBA must execute the START DATABASE command just before making a copy of the database. If there is a backup copy of the database and a transaction log file that begins with the operations executed immediately after the backup was made, the DBA can bring the backup database up to date with the ROLLFORWARD DATABASE statement. This statement recovers the database through the last terminated transaction. The DBA must load the backup database files and execute the ROLLFORWARD DATABASE statement.

After rolling the database forward, the DBA is the only one who has access to the database, since it is left in an EXCLUSIVE MODE. This state allows the DBA to check the database for errors before making it generally available. Logging does not occur during this checking phase. The DBA must close the database when it has been restored correctly.



## Audit Trails

An audit trail is a file that contains a history of all additions, deletions, updates, and manipulations to a database table. An audit trail serves a purpose similar to that of a transaction log: each is used to maintain a record of modifications to a database, and each can be used to update backup copies of a database.

Three audit trail statements are available to protect the integrity of a table:

- CREATE AUDIT**      creates an audit trail for a table.
- DROP AUDIT**      removes the audit trail on a table.
- RECOVER TABLE**   restores a table using the audit trail.

**Creating an Audit Trail.**    Use the CREATE AUDIT statement to create an audit trail file and to begin writing the audit trail. The format is

---

```
create audit for table-name IN "pathname"
```

---

where *table-name* is the name of the table for which you want to create an audit trail file and *pathname* is the full pathname of the audit trail file. The audit trail file should be on a physical device other than the one that holds the data, so that a system failure affecting the device that holds the data does not also damage the audit trail. If your computer system has more than one hard disk, the audit trails could be written to one of the disks not containing the data.

To use the audit trail, make a backup copy of the table *after* you have executed a CREATE AUDIT statement but *before* you have made any changes to the table. Once you have started the audit trail and have made a backup, you are ready to work with the table.

You can drop and create an audit trail file whenever you want. Drop and create the audit trail files just before you make a complete backup of the device containing the data file. If a system failure should occur, you can use the audit trail to back up the table from the time of the last backup to the time the failure occurred.

**Recovering a Table.** In the event of a system failure, you can use the RECOVER TABLE statement to restore a database table by using a backup copy of the table and an audit trail file. You must first restore a backup copy of the table. The backup copy must be in the same state that it was in when the audit trail was started. If it is not in the original state, the recovery fails. The format of the recovery statement is

---

```
recover table table-name
```

---

where *table-name* is the name of the table you want to recover.

Once you recover the table, use the DROP AUDIT statement to remove the contents of the audit trail file. Next, run the CREATE AUDIT statement to start a new audit trail file, then make a new backup copy of the table.

## Comparison of Transactions and Audit Trails

Transactions provide data integrity in two ways. First, they guarantee that SQL statements are either successfully completed or are completely canceled. If, for example, you update several rows of one or more tables within a transaction, the entire update is guaranteed either to succeed by updating all rows, or to fail without changing any rows. Second, the transaction log can be used to recover an entire database.

Audit trails are associated with individual tables. They do not guarantee that modifications to several rows of a table either succeed entirely or fail without any effect. An audit trail file can be used only to recover the table for which it is created.

You should consider using audit trails when you have only a few critical tables and where you do not need the additional facilities of transactions. In this case, the overhead of logging is incurred only



when the critical tables are modified. If you need to maintain the integrity of the database as a whole, or need the guarantee that SQL statements are executed as a unit either entirely or not at all, you must use transactions.

## Locking Statements

**INFORMIX-ESQL/C** automatically locks a record where integrity problems could arise if two or more programs accessed the record simultaneously. Explicit table or record locking is generally not required. **INFORMIX-ESQL/C** allows you to temporarily limit access to a table, however, by executing the **LOCK TABLE** statement. You also can specify whether you want your program to fail if a record it wants is locked, or to wait for the record to become available.

**LOCK TABLE** limits access to the table to the current user only or allows other users to only read the table. Use the **LOCK TABLE** statement only when making major changes to a table in a multi-user environment and when simultaneous interaction with the table by another user would interfere. **LOCK TABLE** decreases the accessibility of the database, since it prevents other users from accessing the table.

**UNLOCK TABLE** restores access to a previously **LOCKED** table.

Ordinarily, when **INFORMIX-ESQL/C** locks a row, other attempts to access the data fail. If you wish to change this strategy, use the following statement:

**SET LOCK MODE** alters the locking strategy either to fail when a row is already locked, or to wait for the lock to be released before proceeding.

See the section "Locking" later in this chapter for a complete discussion of the kinds of locking available and considerations for their use.

# *Dynamic Management*

The preceding discussion assumes that you embed explicit SQL statements in your INFORMIX-ESQL/C program. That is the case for most applications where you are performing predetermined activities on your database. There are several advanced applications, however, where you do not know the statement at compile time. For example:

- Interactive programs, where the user enters a query from the keyboard at run time
- Programs intended to work with different databases whose structure can vary

In situations like these, you must work with dynamically defined statements. The application of dynamically defined statements is more difficult than the use of non-dynamic statements. A brief description of the dynamic management statements follows:

<b>PREPARE</b>	takes a character string, interprets it as one or more SQL statements, and assigns it to a statement identifier. The following dynamic management statements refer to the SQL statement through the statement identifier.
<b>EXECUTE</b>	runs the previously PREPARED statement associated with the statement identifier. Use the EXECUTE statement for non-SELECT statements.
<b>DESCRIBE</b>	determines whether a previously PREPARED statement is a SELECT statement and, if so, gathers information about the storage requirements for a single row. For statements other than SELECT, DESCRIBE returns only the statement type.
<b>DECLARE</b>	declares a cursor for a PREPARED SELECT statement.
<b>FREE</b>	releases all resources associated with a PREPARED statement or an OPENed cursor. The statement must be PREPARED, or the cursor OPENed, before it can be used again.

**EXECUTE** prepares an SQL statement, executes it, then frees all  
**IMMEDIATE** resources associated with the prepared SQL  
statement.

The SQL statement(s) passed to PREPARE at run time cannot refer directly to host variables, since the program has already been compiled. You should insert question marks (?) where you would have put the host variables. These question marks indicate the parameters of the statement(s).

The following example illustrates the use of a dynamically defined non-SELECT statement. Dynamically defined SELECT statements will be described in more detail in Chapter 2 in association with the **sqllda** structure.

---

```
$ char state[128];
char num[12];
long atol( );
$ long p_num;
.
.
.
/* read statement from terminal */
getline(state, 128);

$ prepare stateid from $state;
for(;;)
{
    printf("Enter customer number: ");
    getline(num, 12);
    p_num = atol(num);
    if (p_num == 0)
        break;
$ execute stateid using $p_num;
}
```

---

In this program fragment, an SQL statement is read from the terminal and placed into the character array **state**. It is presumed that the statement contains a single unknown value of type **INTEGER**, such as:



---

```
$ delete from customer where customer_num = ?;
```

---

The program PREPAREs the statement and calls it **stateid**. The next part of the program is an infinite loop that requests personnel numbers and EXECUTEs the statement, substituting the value in the host variable **p\_num** for the question mark in the statement. The loop terminates when you enter a value of zero for the personnel number.

## Locking

INFORMIX-ESQL/C uses locking to prevent different users from executing conflicting operations on the same data. Without locking, for example, two users may be allowed to update the same row at the same time. In this situation, the computer memory contains two different versions of the rows (the one being updated by user A and the one being updated by user B). Without some method of *concurrency control*, the user whose row is the last one actually written to the file “wins” and overwrites the other user’s changes.

INFORMIX-ESQL/C provides two levels of locking to prevent such an occurrence:

- Row-level or record-level locking
- Table-level or file-level locking

INFORMIX-ESQL/C performs *row-level* locking implicitly. The locking strategy may differ slightly, depending upon whether or not the database uses transaction management.

Data definition statements, such as ALTER TABLE, CREATE INDEX, and so on, use implied *table-level* locking. You can explicitly specify table-level locking. The following sections describe the levels of locking and the methods for employing them.

---

INFORMIX-OnLine supports additional functionality. Refer to the *INFORMIX-OnLine Programmer's Manual* for more information.

---



## **Row-Level Locking**

Ordinarily, **INFORMIX-ESQL/C** locks a row when you execute an **UPDATE** statement, or when you execute a **FETCH** statement and the cursor is **DECLARED** with a **FOR UPDATE** clause. If the **UPDATE** statement affects only one row, **INFORMIX-ESQL/C** releases the lock immediately after performing the update. This prevents two programs from attempting to update the same record at the same time. One program receives the lock and can proceed with the update. The other program either fails in its attempt or waits for that program to release the lock. (See the section “Wait for Locked Row” later in this chapter.) If the **UPDATE** statement affects more than one row, **INFORMIX-ESQL/C** uses the same row-locking strategy. As soon as it completes an update, it releases the lock, then locks and updates the next record. When the **UPDATE** finishes, all records are unlocked.

If you want more control over the update of multiple records, you can **DECLARE** a cursor **FOR UPDATE**. The **WHERE** clause of the **SELECT** statement specifies the rows that you want to update. After you **OPEN** the cursor and **FETCH** a record, that record remains locked until you either **CLOSE** the cursor or **FETCH** the next record.

## **Row-Level Locking in Transactions**

If your database uses transaction management, rows that you **INSERT**, **UPDATE**, or **DELETE** within a transaction remain locked until the end of the transaction. The end of a transaction is either a **COMMIT WORK**, where all modifications are made to the database, or a **ROLLBACK WORK**, where no modifications are made.

If you **DECLARE** a cursor **FOR UPDATE**, the **FETCH** statement locks the row exclusively. If your application updates this record with the statement **UPDATE WHERE CURRENT OF**, this lock is held until the **COMMIT WORK** or **ROLLBACK WORK** statement is encountered. If the row is *not* updated, however, then the lock is released upon the next **FETCH**.

## Table-Level Locking

You can use table-level locking to lock an entire table and prevent others from altering or seeing rows in that table.

You may want to use this form of locking during batch operations that affect every row in a table, for example. If the operations must be completed as a single transaction, it may be more efficient to lock the entire table before the transaction begins.

Normally, under transactions, INFORMIX-ESQL/C locks each row as it is UPDATED, DELETED, or INSERTed. If you lock the entire table, however, INFORMIX-ESQL/C does not use row-level locking because it is unnecessary. As a result, you are not likely to reach the limits that your operating system may place on the number of rows that can be locked at any one time. INFORMIX-ESQL/C performs table-level locking automatically as part of the following statements: ALTER TABLE, DROP TABLE, CREATE INDEX, ALTER INDEX, and DROP INDEX.

INFORMIX-ESQL/C locks a table when you execute a LOCK TABLE statement. The syntax for this statement is

---

```
lock table table-name in {share | exclusive} mode
```

---

where *table-name* is the name of the table to be locked.

If you lock the table IN SHARE MODE, other users are able to SELECT data from the table, but they are not able to INSERT, DELETE, or UPDATE rows in the table. If you lock the table IN EXCLUSIVE MODE, other users are not able to access the table at all until you execute an UNLOCK TABLE statement.

Because locking an entire table prevents others from adding or altering data in the table, use this feature sparingly. Lock the entire table only when row-level locking (as described in the previous section) will not suffice.

## ***Wait for Locked Row***

If another user locks a row in a table at the row level and you attempt to alter or delete that row (or examine it with a **SELECT** statement **FOR UPDATE**), **INFORMIX-ESQL/C** returns an error stating that the row is locked. If you prefer that **INFORMIX-ESQL/C** wait on any locked row until the competing process unlocks it, you can execute the **SET LOCK MODE TO WAIT** statement. From then on, your request waits until **INFORMIX-ESQL/C** unlocks the requested row, and you do not receive an error message.

If, for some reason, the competing process fails without unlocking the row, your process deadlocks. Consequently, use the **SET LOCK MODE** statement with caution.

If another user locks a table **IN EXCLUSIVE MODE** and you attempt to alter, delete, or even read a row in the table, **INFORMIX-ESQL/C** returns an error code. The wait-for-lock feature applies only on multi-user systems that support record-level locking.

---

**INFORMIX-OnLine** supports additional functionality. Refer to the ***INFORMIX-OnLine Programmer's Manual*** for more information.

---



# User Status and Privileges

When you create a database, you are automatically the DBA of that database and are the only one who has access to it. Another user cannot have access to a database until you grant the **CONNECT** privilege to that person. Another user cannot create or drop tables and indexes unless granted the **RESOURCE** privilege. Only the Database Administrator (you, initially) can grant these privileges. You also can grant the DBA privilege to another user.

The DBA privilege extends all the powers of the Database Administrator to the grantee, including the ability to alter tables in the system; to drop, start, and roll forward a database; and to grant **CONNECT**, **RESOURCE**, and DBA privileges to others.

If you have the **RESOURCE** privilege, you have the **CONNECT** privilege by default. With the DBA privilege, you have both the **RESOURCE** and **CONNECT** privileges. You can only revoke the privilege of a DBA grantee; you cannot revoke your own DBA privilege. If you, as the creator of a database, grant DBA privileges to another user, that user can revoke the DBA privilege from you, the database creator. This last property permits the transfer of authority from the maker of the database application to the person who has responsibility for maintaining the database.

**INFORMIX-ESQL/C** allows the **CONNECT** and **RESOURCE** privileges to be granted to the **PUBLIC** in addition to specifically named users.

In addition to these database-level privileges, the owner of a table can grant a number of table-level privileges. These permit the grantee access to specific columns to execute **SELECT** or **UPDATE** statements, or give the grantee authority to insert new rows, delete old rows, create indexes, and alter the structure of the table.

In a non-MODE ANSI database, the default is to **GRANT** all table-level permissions (except **ALTER**) to all users (**PUBLIC**). In a MODE ANSI database, no default table-level privileges are granted; you must explicitly grant these permissions. However, if you use **START DATABASE** to convert your database to MODE ANSI, the existing privileges remain in effect unless you specifically **REVOKE** them.

A number of SQL statements can be executed only by the DBA or by the owner of the table or index specified in the statement. These include: **ALTER INDEX**, **ALTER TABLE**, **DROP INDEX**,

DROP TABLE, DROP VIEW, GRANT, RENAME COLUMN, RENAME TABLE, and REVOKE. (You can give others the privilege of executing the ALTER TABLE, GRANT, and REVOKE statements with certain restrictions.)

The owner of a table is the person (login name) who executed the CREATE TABLE statement. The owner of an index is the one who executed the CREATE INDEX statement. Execution occurs when the compiled INFORMIX-ESQL/C program containing the CREATE statements is run, *not* when the INFORMIX-ESQL/C program is compiled.

**Note:** In the INFORMIX-ESQL/C demonstration database, RESOURCE privileges have been granted to PUBLIC.

## Indexing Strategy

There are two major purposes for creating an index on columns of a database table: to speed sorting of rows and to optimize the performance of queries. When your application writes reports involving complex queries through a large database, significant time savings can result from indexing.

The drawback to having an index is that indexes slow down the process of inserting new data into the database. When you update a table, its indexes may also be modified. This is not a problem when you are adding information interactively, a row at a time, but can become time-consuming when it is necessary to insert a large number of rows from one table into another.

The solution to this potential conflict between needs is to take a pragmatic approach to indexing. One of the advantages of an Informix database is that you do not have to decide issues such as which columns to index at the time that you create your tables. You should write your applications to create indexes when you need them and to drop them when they get in the way.

It takes time to create an index on a table already containing data, so create only those indexes that optimize the queries you make. For example, you may be able to schedule creating your indexes in anticipation of batch report writing during the night and drop them the next morning before there are huge data-entry needs.



The following suggestions are hints for effective indexing. Although the last two refer to a single query, they apply when you anticipate making a number of queries with the same qualities.

- Do not create indexes for small tables with fewer than 200 rows. The speed you gain from using an index does not overcome the time required to open and search the index file on small tables.
- Do not create indexes on a column that has only a few possible values. Such columns are those that contain data like sex, marital status, yes/no responses, or zip codes in a small city. Performance of queries using these columns is not likely to improve and performance of inserts, deletes, and updates is likely to be slower.

If you frequently need to have data sorted on columns with a small range of possible values, create a temporary table of the sorted data. Another approach is to redesign the database with separate tables for each alternative value.

- If the WHERE clause of a SELECT statement imposes a condition on a single column, put an index on that column. If there are conditions placed on several columns, make a composite index on all the affected columns. For the SELECT statement

---

```
$ select * from items where order_num > 1015;
```

---

put an index on **order\_num**. For the statement

---

```
$ select * from items
    where order_num = 1015
    and total_price > 1000.00;
```

---

create a composite index on both **order\_num** and **total\_price**.

- If the WHERE clause of a SELECT statement has a join condition between a single column in one table and a single column in another table, create an index on the column in the table with the larger number of rows. If several columns of one table have join conditions with several columns in another table, create a composite index on the affected columns of the table with the larger number of rows.



## For the SELECT statement

---

```
$ select * from items, stock
      where items.stock_num = stock.stock_num;
```

---

place an index on **stock\_num** in the **items** table, since it has many more rows than the **stock** table. Execute the UPDATE STATISTICS statement before the SELECT statement so that SQL knows the current size of the tables.

## For the statement

---

```
$ select * from items, stock
      where items.stock_num = stock.stock_num
      and items.manu_code = stock.manu_code;
```

---

put a composite index on **stock\_num** and **manu\_code** in the **items** table.

It is not always easy to know how indexes are used during a query, but you can determine this by issuing the SET EXPLAIN statement. When you set this statement to ON, a file called **sqexplain.out** is created in the current directory. A description of the decisions made by the *query optimizer*, a feature of the engine used to improve performance, is written into this file for each subsequent query. The recorded information includes the order of table access, how filters are applied, and what (if any) indexes are used in processing the query.

For example, if your queries seem to be taking longer than necessary, you may choose to change your indexing strategy. However, in a complex query, it may be difficult to predict the actual order of actions taken by the optimizer, thus making it difficult to determine what (if any) indexes should be added or dropped. The SET EXPLAIN statement provides you with information to determine exactly how the database is being accessed and to help you assess whether changing indexes may improve the decisions of the optimizer.

## Auto-Indexing

If you execute a **SELECT** statement that includes a join between two tables and there are no indexes on the joined columns, **INFORMIX-ESQL/C** creates a temporary index on the table with the larger number of rows before performing the join. The index disappears when the query finishes. This enhancement is transparent to the user except for a dramatic improvement in the speed of unindexed queries.

## Clustered Indexes

Since **UNIX** extracts information from the disk in blocks, the more rows that are physically on the same block and are already in the order of an index, the faster sequential retrieval using an index will proceed. Ordinarily, no relationship need exist between the physical order of the data in the **.dat** file and the order of an index. You can, at least temporarily, cause the physical order in the table to be the same as the order in an index through *clustering*.

**INFORMIX-ESQL/C** clusters (orders) the physical data in a table when you create a new index by executing the **CREATE CLUSTER INDEX** statement, or when you execute the **ALTER INDEX** statement to cluster for an existing index. Since users who have access to the table can add rows or update the information in existing rows, a table that you cluster according to an index does not stay that way. Over time, you can expect the benefit of an earlier clustering to disappear and you may want to cluster the table again using an **ALTER INDEX TO CLUSTER** statement.

Since a table can have only one physical order, you can have only one clustered index on a table at any given time. You can change the physical order to reflect a different index by executing two **ALTER INDEX** statements:

- Execute an **ALTER INDEX TO NOT CLUSTER** statement to release the cluster attribute from the first index.
- Execute an **ALTER INDEX TO CLUSTER** statement to attach the cluster attribute to the second index.

**Note:** You cannot execute an **ALTER INDEX** or **CREATE INDEX** statement on a view.

## NULL Values

The basic purpose of introducing NULL values in a database is to indicate when no value has been assigned to a particular column in a particular row of a table. Your reasons for not having assigned a value could either include not knowing the correct value or that no value yet exists. The NULL may also indicate that no value is appropriate for a given column because of the values that were entered into other columns.

For example, consider entering data for a bank customer who is requesting a loan. If the customer, Mr. Farthing, is not employed, the **employer** column in the **client** table has no entry for this customer. This CHAR column has the value NULL. The **hire\_date** column is meaningless if Mr. Farthing is not employed. There is no appropriate date to enter; the value is NULL.

## Default Values

In INFORMIX-ESQL/C, the default value for a column is NULL. INFORMIX-ESQL/C makes a distinction between zero and NULL for numeric values and between blanks and NULL for character values.

By definition, type SERIAL columns can never contain the NULL value. Columns of type SERIAL always contain integers greater than or equal to one.

You can insist that a column of any type not have NULL values by using the NOT NULL clause in the CREATE TABLE statement. INFORMIX-ESQL/C prevents a NULL from being entered into any column that is declared NOT NULL. You cannot, however, use a NOT NULL clause in an ALTER TABLE statement when you add a new column. The reason is that INFORMIX-ESQL/C enters a NULL value into that column for all rows that already exist.

A column for which you create a unique index can have, at most, one NULL value.



## Note for users of the Version 1.10 implementation of SQL:

In Version 1.10 SQL, when no value is provided for a column entry in a row of a table, INFORMIX-ESQL/C enters a blank for type CHAR columns, zeroes for number columns, and a very large negative value for type DATE columns. Since zero could well be an acceptable value for a number column (for example, the value for a type MONEY column), there is no way to distinguish an unknown value from zero.

To incorporate your existing Version 1.10 database into INFORMIX-ESQL/C programs, you must execute the **dbupdate** utility described in Appendix F. (Appendix F also describes how you can avoid using NULL values.)

## *The NULL in Expressions*

If any value that participates in an arithmetic expression is NULL, the value of the entire expression is NULL. For example, consider the following query:

---

```
$ select order_num, ship_charge/ship_weight
   from orders
   where order_num = 1023;
```

---

When **ship\_charge** is NULL because the order with number 1023 is new and the shipping charge has not yet been determined, the value returned for **ship\_charge/ship\_weight** is also NULL.

The situation is different when you use one of the aggregate functions. (See Chapter 3 for a description of aggregate functions.) COUNT(\*) counts all rows, even if the value of every column in the row is NULL. COUNT(DISTINCT *column-name*), AVG, SUM, MAX, and MIN ignore rows with NULL values for the column in their argument and return the appropriate value based on the rest of the rows. However, if a column contains only NULL values, then COUNT(DISTINCT *column-name*) returns zero, and the other four aggregate functions return NULL for that column.

## The NULL in Boolean Expressions

In order to incorporate NULL values into Boolean expressions, it is necessary to enlarge the number of truth values from simply true and false to include unknown. If one of the expressions of a Boolean expression is NULL, the truth value of the Boolean expression is unknown. For example, the Boolean expression

---

`ship_charge/ship_weight < 5.0`

---

has the truth value unknown for the order in the previous example.

If you combine Boolean expressions using the operators AND, OR, and NOT, the following tables give the resulting truth value (where T corresponds to true, F to false, and ? to unknown).

AND	T	F	?	OR	T	F	?	NOT	
T	T	F	?	T	T	T	T	T	F
F	F	F	F	F	T	F	?	F	T
?	?	F	?	?	T	?	?	?	?

## The NULL in WHERE Clauses

If the Boolean expression in a WHERE clause evaluates to unknown for a particular row, INFORMIX-ESQL/C treats the search condition as not satisfied and does not select or modify that row.

Consider the following clause:

---

`where ship_charge/ship_weight < 5`  
`and order_num = 1023`

---

The row where `order_num = 1023` is the row where `ship_charge` is NULL. Since `ship_charge` is NULL, `ship_charge/ship_weight` is also NULL, and the truth value of `ship_charge/ship_weight < 5` is unknown. Since `order_num = 1023` is true, the preceding AND truth table states that the truth value of the entire search condition is unknown. Consequently, that row is not chosen. If the search

condition had used an OR in place of the AND, the search condition would be true.

You can select (or reject) rows containing NULL values with a new type of search condition:

---

*column* IS [NOT] NULL

---

You must use the keyword IS. It is not permitted to write the condition as follows:

---

<i>column</i> = NULL	(Incorrect)
<i>column</i> != NULL	(Incorrect)

---

If you perform a join between two tables using the WHERE clause,

---

where *column1* = *column2*

---

INFORMIX-ESQL/C does not select the rows where either **column1** or **column2** is NULL. In particular, no row is returned if both **column1** and **column2** are NULL. This is merely a special case of the more general rule that Boolean expressions containing NULL values have an unknown truth value.

Similarly, if a subquery returns a single NULL value, the search condition evaluates to unknown.

## ***The NULL in ORDER BY Clauses***

For the purpose of sorting rows using the ORDER BY clause, the NULL value is treated as being less than a non-NULL value. When the ordering is ASC, the NULL values come first; when the ordering is DESC, the NULL values come last.



## *The NULL in GROUP BY Clauses*

INFORMIX-ESQL/C treats each row containing a NULL value in the column being GROUPed BY as part of a single group.

## *The NULL Keyword in INSERT and UPDATE Statements*

When you execute the INSERT statement, INFORMIX-ESQL/C inserts the NULL value into all columns for which you do not provide a value and for all columns not listed explicitly. Since the *value-list* of the INSERT statement must be the same length as the *column-list*, you can use the keyword NULL to indicate that a column in *column-list* should be assigned a NULL value.

---

```
$ insert into orders (order_num, order_date,  
                      customer_num) values (0, null, 123);
```

---

All other columns in the **orders** table are filled with NULL values. Similarly, you can use the NULL keyword to modify a column value when using the UPDATE statement. For a customer whose previous address required two address lines, but now requires only one, you would use the following entry:

---

```
$ update customer  
  set address1 = "123 New Street",  
      address2 = null,  
      city = "Palo Alto",  
      zipcode = "94303"  
  where customer_num = 134;
```

---

# Views

Views are constructs on a database that allow you to do the following tasks:

- Provide different users with different windows (called “views”) on the data in the database. A single view may involve columns from different tables or may show values that are functions of the values from the columns. A view has a name and looks to a user as if it were a table. The user can query a view, for example, using the same syntax as though the view were a table in the database.
- Limit access to sensitive data by allowing users to see only aggregate information. With the GRANT and REVOKE statements, you can prevent a user from seeing any salary data in a personnel table. With a view, you can allow the user to see average salaries in various groups, but still protect the individual salary data.
- Permit users to update, insert, and delete data in the database as though the data were organized as it appears in a view. You can also examine the changes made in a real table of the database through a view.

Views are therefore dynamic windows into the database and are not static snapshots. They differ in this respect from a temporary table created by the INTO TEMP clause of a SELECT statement or the CREATE TEMP TABLE statement. Such temporary tables show you only the state of the database when the temporary table was created.

Although views appear to be tables in the database, there are a few important ways in which they differ.

- You cannot create an index on a view.
- Under certain conditions, you cannot update or modify the data perceived through a view. An obvious case occurs when the “column” seen in a view is really an expression generated from actual database tables. Generally speaking, there is no way to determine the appropriate change in the underlying columns involved in such an expression if you want to change the value of the “column.”

The next sections describe how to create and delete views, how to query the database through views, how to modify the database through a view, and how to set up permissions for a view.

## *Creating and Deleting Views*

You must use the **CREATE VIEW** statement to create a view. As described in Chapter 3, a view is determined by a **SELECT** statement that returns the “table” that defines the view. You cannot use the **UNION** operator (see Chapter 3 for the definition of the **UNION** operator) in the definition of a view. The **SELECT** statement is stored in the **sysviews** system catalog. When you subsequently make reference to a view in another statement, **INFORMIX-ESQL/C** performs the defining **SELECT** statement in executing the new statement.

You can use the same column names as in the underlying table for the view or you can assign new names. When a column in a view is the evaluation of an expression or is not unique (because, for example, you have included all the columns of a join, including the columns that define the join), you must supply new names. These column names are stored in **syscolumns** along with the column names of regular tables.

You can delete a view by executing the **DROP VIEW** statement. When you drop a view, you also drop all views that were defined in terms of that view.

## *Querying through Views*

You can make queries involving views exactly as though they were tables in the database. If possible, **INFORMIX-ESQL/C** first combines the view-defining **SELECT** statement with the query to create a new **SELECT** statement and then executes the new statement. Otherwise, it creates the view as a temporary table and applies the query to the table. **INFORMIX-ESQL/C** may detect errors during either of these phases and return a negative value in the structure element **sqlca.sqlcode**. (See Chapter 2 for a discussion of the **sqlca** structure.)



## *Modifying through Views*

As in querying through views, you can use the INSERT, UPDATE, and DELETE statements with views. INFORMIX-ESQL/C combines the view-defining SELECT statement with the view-referring statement and then executes it. The following restrictions apply to modifying tables through a view:

- You cannot modify the database through a view if the view definition involves joins, the GROUP BY clause, the UNIQUE keyword, or an aggregate function. If any of these features is present in the view definition, you cannot execute INSERT, DELETE, or UPDATE statements on the view. You can, however, define a view using a subquery that refers to another table and can often circumvent the restriction on joins (see the section “Data Constraints Using Views” later in this chapter).
- A view column can be UPDATED only if it is derived directly from a table of the database and not as a result of an expression. Expression-derived columns are called “virtual” columns. You cannot INSERT rows through a view that contains virtual columns, although you can DELETE a row that contains a virtual column.
- You cannot execute the ALTER TABLE, CREATE INDEX, or UPDATE STATISTICS statements on a view. You do receive the benefit of existing indexes on the underlying tables.

You can use an INSERT statement on a view that shows only a portion of an underlying table. When you do so, the unmentioned columns of the underlying table receive NULL values. If one of the unmentioned columns does not permit NULL values, INFORMIX-ESQL/C does not permit you to INSERT to the view.

If you drop a column of an underlying table of a view that you have defined in terms of that column, INFORMIX-ESQL/C issues an error if you subsequently make reference (other than DROP VIEW) to the view.

Unless you create the view with a WITH CHECK OPTION clause, it is possible to INSERT or UPDATE data in a database through a view that does not satisfy the limitations on the view. A row INSERTed or UPDATED in this manner is no longer accessible through the view.

For example, a view could be created that allows the user access only to customers from Palo Alto. If, when using the view, the user creates a new row with a customer from Menlo Park, the user cannot select the row through the view. If the `city` column on an existing row is `UPDATED` to Menlo Park, the row disappears from the view. The `WITH CHECK OPTION` clause on the `CREATE VIEW` statement causes `INFORMIX-ESQL/C` to reject an `UPDATE` or `INSERT` that violates the restrictions of the view.

You must be careful when you `UPDATE` a table through a view that may contain duplicate rows. Duplicate rows can occur in a view even if the underlying table has unique rows. If a view is defined on the `items` table and contains only the columns `order_num` and `total_price`, the view contains duplicate rows if two items from the same order have the same total price. If you put the cursor on one of the rows where `total_price` = \$1234.56 and update the `total_price` to \$1250.00 through the view, you have no way of knowing which item you have increased.

## *Privileges with Views*

When you create a view, you receive the same privileges that you had on the underlying tables. If you have these privileges with the `WITH GRANT OPTION` (see Chapter 3), you can grant privileges on your view to other users.

If the view is built on more than one table, you can have only the `SELECT` privilege since multi-table views do not permit you to `INSERT`, `DELETE`, or `UPDATE`. You must have the `SELECT` privilege on all the columns from which a multi-table view is derived to have the `SELECT` privilege on the entire view. If, as a result of these restrictions, you have no privileges on a view, the `CREATE VIEW` statement returns an error code.

## *Data Constraints Using Views*

The purpose of data constraints is to ensure that all data entered into the database satisfies preassigned limitations. It is often desirable to define allowed value ranges dynamically, based on the values in other columns or even in other tables. The existence of views and, specifically, the **WITH CHECK OPTION** clause permits the DBA to control the entry of data into the database. This is most easily demonstrated with an example taken from the **stores** database.

Suppose you want to ensure that no item

- Has a value of more than \$20,000
- Is for stock that does not exist

The first step is to create the following view:

---

```
$ create view safe_items as
  select * from items
    where total_price < 20000 and
      exists (select stock_num, manu_code
              from stock
              where stock.stock_num
                 = items.stock_num
                 and stock.manu_code
                 = items.manu_code)

  with check option;
```

---

If you do all data entry and data modification through the **safe\_items** view, **INFORMIX-ESQL/C** rejects all data that does not meet the requirements of the **WHERE** clause. Because of the dynamic nature of views, the view will only contain rows corresponding to current stock items if you change the **stock** table by adding rows corresponding to new stock items or by deleting old ones.

By extending the **WHERE** clause, this example can be expanded to cover very general data-constraint needs.



## Outer Joins

An outer join between two tables treats the two tables unsymmetrically. One of the tables is dominant (often referred to as “preserved”), and the other table is subservient. If the subservient table has no rows satisfying the join condition, the outer join attaches a row of NULL values to the row of the dominant table before projecting the desired columns. To illustrate, let **a** be a column from **tab1** and **b** a column in **tab2**. Further, let the values in the two tables be as shown in the following display:

tab1.a	tab2.b
2	4
3	2
5	6
	5

INFORMIX-ESQL/C syntax requires that the subservient table in an outer join be preceded by the keyword **OUTER** in the **FROM** clause. The following **SELECT** statement contains an outer join between **tab1** (the dominant table) and **tab2** (the subservient table):

---

```
$ select a, b
      from tab1, outer tab2
      where a = b;
```

---

The resulting table has the following two columns:

a	b
2	2
3	-
5	5

Every value for **a** is present, and only those values of **b** that match those in **a** are present. When there is no value in column **b** that satisfies the join condition, a NULL value (shown here as -) is substituted.

A WHERE clause is required in the case of outer joins and must set a condition between the two tables.

See Chapter 3 and Appendix G for more information about outer joins.

## Table Access by ROWID

The keyword ROWID can be used in INFORMIX-ESQL/C statements to refer to the internal record number associated with a row in a database table. ROWID can be thought of as a hidden column in every table. When you refer to *table.\**, the implied list of columns does *not* include ROWID. On the other hand, you can use the syntax

---

```
select rowid, * from table
```

---

to get the ROWID value for each row. You can also determine the ROWID of the last row that INFORMIX-ESQL/C dealt with by examining the *sqlca* structure (see Chapter 2).

ROWID can also be used in WHERE clauses to select rows based on their internal record number. This feature is useful when there is no other unique column in a table.

If a row is deleted from the table, its ROWID may be assigned to a new row. You should not attribute chronological or other significance to the sequential values of ROWID.

# TODAY, CURRENT, and USER Functions

INFORMIX-ESQL/C provides functions to allow you to include the date, the date and time of day, and the login name of the current user in an SQL statement.

- TODAY always returns the system date.
- CURRENT returns the system date and time.
- USER returns a string containing the login account name of the current user.

You can use these functions wherever you use a constant. For example, if you want to retrieve the rows that you have inserted into a table, you must first define a CHAR column to contain the USER name.

When you insert new rows into the table, use the USER function as follows:

---

```
$ insert into table values (... ,user,...);
```

---

You can then retrieve the rows you entered with a SELECT statement, as follows:

---

```
$ select * from table
    where user_col = user;
```

---

(See the section "Cursor Management" earlier in this chapter for a discussion on using the SELECT statement to return multiple rows.)

Use the TODAY function in the same way. You can insert the system date into a table with the following statement:

---

```
$ insert into table values (... ,today,...);
```

---



The next statement retrieves all rows with the current date from *table*:

---

```
$ select * from table
  where date_col = today;
```

---

You can use CURRENT to insert the system date and time:

---

```
$ insert into table values (... ,current,...);
```

---

The next query selects rows whose DATETIME value is within a range from the beginning of 1989 to the current instant:

---

```
$ select * from table
  where dt_col between "1989-1-1 00:00:00" and current;
```

---

## **Chapter 2**

# **C Programming Using Embedded SQL**

© 1999-2000

Copyright 1999-2000  
All rights reserved.



## Chapter 2 Table of Contents

Chapter Overview .....	5
Header Files .....	6
Include Files .....	7
Host Variables .....	8
Indicator Variables .....	14
Embedding SQL Statements in C Routines .....	17
Error Handling and the <i>sqlca</i> Structure .....	18
Dynamic SQL Statements and the <i>sqlda</i> Structure .....	21
The <i>sqlda</i> Structure .....	21
Non-Parameterized SELECT Statements .....	23
Parameterized SELECT Statements .....	27
Using Host Variables .....	27
Using the <i>sqlda</i> Structure .....	28
Parameterized Non-SELECT Statements .....	29
The ESQL/C Preprocessor .....	29
Preprocessor Support .....	29
Compiling ESQL/C Routines .....	31
demo1.ec .....	35
demo2.ec .....	36
demo3.ec .....	38
unload.ec .....	40



# Chapter Overview

This chapter describes the structure of an **INFORMIX-ESQL/C** program and introduces several basic concepts. You will learn

- How SQL statements are embedded in C programs
- How C variables are identified with database constructs
- How to detect errors
- How to keep track of queries that return multiple rows
- How to use dynamically defined SQL statements
- How to preprocess and then compile your C program

C programs that use **INFORMIX-ESQL/C** statements generally include the following elements:

- Header files
- Include files
- Host variables
- Indicator variables
- **INFORMIX-ESQL/C** statements
- Error handling

Your program also may include dynamically defined statements. This chapter provides details on each of these topics.



# Header Files

The C program that contains **INFORMIX-ESQL/C** statements may include the following header files:

<b>sqlca.h</b>	contains the structure in which error messages are stored.
<b>sqllda.h</b>	contains the structures that contain value pointers and descriptions of dynamically defined variables.
<b>sqlstype.h</b>	contains integer constants corresponding to SQL statements.
<b>sqltypes.h</b>	contains the definitions of strings corresponding to C language and SQL data types.

The syntax for including these files is as follows:

---

```
$include    sqlca;  
$include    sqllda;  
$include    sqlstype;  
$include    sqltypes;
```

---

Always include **sqlca.h** to check the success or failure of your **INFORMIX-ESQL/C** statements. You need not include **sqllda.h**, **sqlstype.h**, or **sqltypes.h** unless your program makes reference to the structures or the definitions included in them.

You can read more about **sqlca.h** and **sqllda.h** later in this chapter. You will find **sqltypes.h** mentioned in association with the **DESCRIBE** statement in Chapter 3, "SQL Statements," while **sqltypes.h** is discussed in Chapter 4, "SQL Data Types." These four header files plus **decimal.h**, **datetime.h**, and **ctools.h** are reproduced in Appendix A.

---

**INFORMIX-OnLine** supports additional functionality. Refer to the *INFORMIX-OnLine Programmer's Manual* for more information.

---

## Include Files

You can include other INFORMIX-ESQL/C files in your program in addition to header files. You do this by using the preprocessor statement **\$include** or **EXEC SQL include**. Use one of the following formats:

---

**\$include** *filename*;

**EXEC SQL include** *filename*;

---

**Note:** Use of the EXEC SQL keywords (in place of the dollar sign (\$)) conforms to ANSI standards.

The INFORMIX-ESQL/C preprocessor reads *filename* into the current file at the position of the include statement, processes it, and passes it on to the C compiler. The standard **#include** of C includes the file after the INFORMIX-ESQL/C preprocessor stage. You must use **\$include** or **EXEC SQL include** if *filename* contains SQL statements.

See the section "Preprocessor Support" later in this chapter for more information on **\$include** statements.

# Host Variables

*Host variables* are normal C variables that you use in SQL statements. When you use a host variable in an SQL statement, prefix its name with a dollar sign (\$) or a colon (:). The host variable **hostvar**, for example, appears in an SQL statement as **\$hostvar** or **:hostvar**.

**Note:** Use of the colon (:) as a host variable prefix conforms to ANSI standards.

Host variables are declared as ordinary C variables, except that the declaration must either be prefaced by a dollar sign (\$) or be contained within an EXEC SQL BEGIN DECLARE SECTION / EXEC SQL END DECLARE SECTION. Examples of both follow.

## Examples

---

```
    /* pointer to a character */
$   char    *hostvar;

    /* integer */
$   int      hostint;

    /* long integer */
$   long     hostlong;

    /* double */
$   double   hostdbl;

    /* character array */
$   char     hostarr[80];

    /* structure */
$   struct {
    int svar1;
    int svar2;
    ...
  } hoststruct;
```

---



---

```
EXEC SQL BEGIN DECLARE SECTION
    char    *hostvar;
    int     hostint;
    long    hostlong;
    double  hostdbl;
    char    hostarr[80];
EXEC SQL END DECLARE SECTION

EXEC SQL BEGIN DECLARE SECTION
    struct {
        int svar1;
        int svar2;
        ...
    } hoststruct;
EXEC SQL END DECLARE SECTION
```

---

**Note:** Use of the EXEC SQL keywords conforms to ANSI standards.

**INFORMIX-ESQL/C** allows you to declare host variables with normal C initializer expressions. However, initializers containing character strings cannot contain embedded semicolons or **INFORMIX-ESQL/C** keywords. Following are valid examples of C initializers.

---

```
$int varname = 12;
$long cust_nos[8] = (0,0,0,0,0,0,0,9999);
```

---

Initializers are not checked for valid C syntax; they are simply copied to the output. The C compiler will diagnose any errors.

Structures can be declared as **INFORMIX-ESQL/C** host objects. In **INFORMIX-ESQL/C** statements, you can name the structure variable as a whole or as its individual components. If a structure name is used, it is expanded into a list of component names. Structures can be nested.

---

```
$struct customer_t
{
    int      c_no;
    char     fname[32];
    char     lname[32];
} cust_rec;
$struct customer_t cust2_rec;
```

---

With the above declaration,

---

```
$insert $cust_rec into customer;
```

---

is equivalent to:

---

```
$insert into customer
values ($cust_rec.c_no, $cust_rec.fname,
       $cust_rec.lname);
```

---

**INFORMIX-ESQL/C** supports standard C **typedef** expressions and allows their use as host variables. For example:

---

```
$typedef short smallint;
$typedef long serial;
$serial row_nums [MAXROWS];
$smallint counter;
```

---

Since host variables appear in SQL statements, they are associated with an SQL data type. In addition, a host variable must be declared as a C data type. The relationship between SQL data types and C data types is described in detail in Chapter 4. The example that follows summarizes the relationship.

---

SQL	C Language
CHAR(n) CHARACTER(n)	char [n + 1]
SMALLINT	short int
INTEGER INT	long int
DECIMAL DEC NUMERIC	dec_t or struct decimal
SMALLFLOAT REAL	float
FLOAT DOUBLE PRECISION	double
MONEY	dec_t or struct decimal
SERIAL	long int
DATE	long int
DATETIME	dtime_t or struct dtime
INTERVAL	intrvl_t or struct intrvl

---

If the host variable is not declared according to this example, INFORMIX-ESQL/C tries to convert data types, if the conversion is meaningful. See Chapter 4 for a discussion of data conversion.

INFORMIX-ESQL/C understands and supports the declaration of arrays of variables. You can use elements of an array within INFORMIX-ESQL/C statements. However, for types other than CHAR, you can *not* use the array name alone. For example, if you declare

---

```
$long customer_nos[10];
```

---



the following is possible:

---

```
for (i=1; i<10; i++)
{
    $fetch customer_cursor into $customer_nos[i];
}
```

---

You can define as many host variables as you need (up to the limit set for the symbol table of your C compiler).

The representation of NULL values depends on both the machine and the data type. Often the representation does not correspond to a legal value for the C data type, and you should not attempt to perform arithmetic or other operations on a host variable that may have a NULL value.

INFORMIX-ESQL/C provides a function that enables you to test whether a host variable corresponds to a NULL value (**risnull**), and a function to set a host variable to a NULL value (**rsetnull**). See Chapter 5 for a description of these functions. You also can define indicator variables for host variables that correspond to database columns that allow NULL values.

The INFORMIX-ESQL/C preprocessor issues a warning if you declare a host variable in one function and then redeclare it in another function. It considers a structure to be redeclared if any of its elements differ in name, type, or length from those declared earlier. The warning states that the preprocessor will use the second declaration from then on. This situation can happen with the declaration of an index **i** as **int** in one function and as **long** in another. If each such variable is a local variable, you can ignore the warning. When you declare the variable as **extern** or as global to a file, however, subsequent redeclaration inside a function can cause subtle errors.

You can assure that the host variables declared within a function are local to that function by using the combined symbol pair `${` and `$}` to open and close the function:

---

```
$  extern int hostvar;
...
func1( )
${
$  long hostvar;
...
}$
...
func2( )
{
...
$  declare q_stock cursor for
      select stock_num
            into $hostvar
            from stock;
...
}
```

---

The host variable **hostvar** declared in **func1** has a scope only within that function. The variable **hostvar** used within **func2** is the one that is declared as **extern**.

You can also use the combined symbol pair `${` and `$}` to open and close blocks within a function. Host variables declared within such a block are local to that block. You can nest blocks up to 16 levels. The global level counts as level one.

**Note:** ANSI standard syntax does not support the `${` and `$}` symbols.

## Indicator Variables

Since a NULL value is often not a definite value among other values, you must be able to find out if an INFORMIX-ESQL/C statement returns a NULL variable to a host variable. If a host variable corresponds to a database column that allows NULL values, you should define an *indicator variable* in association with a host variable. The associated host variable is called a *main variable*.

When an INFORMIX-ESQL/C statement returns a NULL value to a host variable (through the INTO clause of a SELECT or FETCH statement) and you have defined an indicator variable, the indicator variable has a value of -1. The actual value in the host variable may not be a meaningful C value. If you have not assigned an indicator variable to the host variable and a NULL value is returned, INFORMIX-ESQL/C may or may not generate an error depending on how you compiled the program.

- If you compile the program using the **-icheck** flag, INFORMIX-ESQL/C generates an error and sets **sqlca.sqlcode** to a negative value when a NULL value is returned and no indicator variable is present. (See the section “Error Handling and the **sqlca** Structure” later in this chapter.)
- If you compile the program without using the **-icheck** flag, INFORMIX-ESQL/C does not generate an error when a NULL value is returned and no indicator variable is present.

When a non-NULL SQL value is retrieved into a host variable character array, it may have to be truncated to fit. In this case, INFORMIX-ESQL/C sets the associated indicator variable equal to the size in bytes of the SQL variable before truncation. The fact of truncation is signaled in the **sqlca** structure; the indicator variable tells you the extent of the truncation. If the returned value is neither NULL nor truncated, the indicator variable has the value 0.



You specify an indicator variable in an SQL statement in one of two ways:

- Place a colon (:) between the main (host) variable name and the indicator variable name. (You can use a dollar sign (\$) instead of a colon, but the colon makes the code easier to read.) Use the following format:

---

```
$hostvar:hostvarind
```

```
$hostvar$hostvarind
```

---

- Place the INDICATOR keyword between the main variable name and the indicator variable name. Use the following format:

---

```
$hostvar INDICATOR hostvarind
```

---

**Note:** Use of the INDICATOR keyword conforms to ANSI standards.

Use commas to separate variables in host variable lists.

## Examples

---

```
$ char name[16];
$ char comp[20];
$ short nameind;
$ short compind;
.
.
.
$ select lname, company
   into $name:nameind, $comp:compind
   from customer
   where customer_num = 105;
```

---

---

```
EXEC SQL BEGIN DECLARE SECTION
    char    name[16];
    char    comp[20];
    short   nameind;
    short   compind;
EXEC SQL END DECLARE SECTION
.
.
EXEC SQL
    select lname, company
        into $name:nameind, $comp:compind
        from customer
        where customer_num = 105;
```

---

If **lname** is defined in the **customer** table as having a length longer than 15 characters, **nameind** contains the actual length of the **lname** column. **name** contains the first 15 characters (the string **name** must be terminated with a null character). If the last name of the company representative with **customer number = 105** is shorter than 15 characters, only trailing blanks are truncated.

If **company** has a NULL value for this same customer, **compind** has a negative value. The contents of character array **comp** cannot be predicted.

As an alternative to using the NULL keyword in an INSERT statement, you can use a host variable with a negative indicator variable.

**Note:** Prior to Version 4.0, you had to declare an indicator variable as a short integer. Now, with INFORMIX-SE, indicator variables can be of any valid host variable data type *except* DATETIME or INTERVAL. If you include non-short indicators in your code, you should recompile and relink your INFORMIX-ESQL/C programs to avoid errors.

---

INFORMIX-OnLine supports additional data types. Refer to the *INFORMIX-OnLine Programmer's Manual* for more information.

---

# Embedding SQL Statements in C Routines

SQL statements are embedded in a C routine using the dollar sign (\$), or the EXEC SQL keywords. Use either of the following conventions to embed *SQL-statement* in a C routine:

---

```
$ SQL-statement;
```

```
EXEC SQL SQL-statement;
```

---

**Note:** Use of the EXEC SQL keywords (in place of the dollar sign (\$)) conforms to ANSI standards.

**INFORMIX-ESQL/C** statements can include host variables (and usually indicator variables as well) in any place a constant can be used. See the rules for individual statements in Chapter 3 for any exceptions.

You can use a double dash (--) comment indicator on any **INFORMIX-ESQL/C** line prefaced by \$ or EXEC SQL and terminated by a semi-colon. The comment continues to the end of the line. For example:

---

```
$database db;
```

```
-- database statement comment
```

---



## Error Handling and the *sqlca* Structure

Proper database management requires that all logical sequences of statements that modify the database continue successfully to completion. If, for example, you UPDATE a customer account to show a reduction of \$100 in the payable balance and, for some reason, the next step to UPDATE the cash balance fails, your books will be out of balance. It is important to check that every SQL statement executes as you anticipated.

INFORMIX-ESQL/C returns a result code into the *sqlca* structure after executing every SQL statement except DECLARE. This structure is defined in *sqlca.h* and is shown here.

---

```
struct sqlca_s {
    long sqlcode;
    char sqlerrm[72];
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn0;
    char sqlwarn1;
    char sqlwarn2;
    char sqlwarn3;
    char sqlwarn4;
    char sqlwarn5;
    char sqlwarn6;
    char sqlwarn7;
    struct sqlcaw_s {
        } sqlwarn;
} sqlca;
```

---

### Explanation

**sqlcode** indicates the result of executing an INFORMIX-ESQL/C statement.

It is set to zero for a successful execution of most statements and to SQLNOTFOUND (defined as 100 in *sqlca.h*) for a successfully executed query that returns zero rows or for a FETCH that seeks beyond the end of an active set.

**sqlcode** is negative for an unsuccessful execution. See the "Error Messages" chapter following the appendixes for the error codes and their meaning.

**sqlerrm** is not used at present.

**sqlerrp** is not used at present.

**sqlerrd** is an array of six long integers:

**sqlerrd[0]** is not used at present.

**sqlerrd[1]** is a SERIAL value returned after INSERT or an ISAM error code.

**sqlerrd[2]** is the number of rows processed.

**sqlerrd[3]** is not used at present.

**sqlerrd[4]** is the offset of error into the SQL statement.

**sqlerrd[5]** is the ROWID of last row after INSERT.

**sqlwarn** is a structure containing eight characters that signal various warning conditions (as opposed to errors) following the execution of an SQL statement. The characters are blank if no problems were detected.

**sqlwarn0** is set to W if one or more of the other warning characters has been set to W. If **sqlwarn0** is blank, you do not have to check the remaining warning characters.

**sqlwarn1** is set to W if one or more data items was truncated to fit into a character host variable. You can discover which item was truncated by examining the associated indicator variables. **sqlwarn1** is set to W following a DATABASE statement if the database has transactions.

**sqlwarn2** is set to W if an aggregate function (SUM, AVG, MAX, MIN) encountered a NULL value in its evaluation. **sqlwarn2** is set to W following a DATABASE statement if a MODE ANSI database (with transactions) is selected.

**sqlwarn3** is set to **W** when the number of items in the *select-list* of a **SELECT** clause is not the same as the number of host variables in the **INTO** clause. The number of items returned by **SQL** never exceeds the number of host variables. **sqlwarn3** is set to **W** following a **DATABASE** statement if an **INFORMIX-OnLine** database is selected.

**sqlwarn4** is set to **W** by the **DESCRIBE** statement when an **UPDATE** or **DELETE** statement is **PREPARED** without a **WHERE** clause. Without a **WHERE** clause, the **UPDATE** or **DELETE** applies to the entire table. By checking this variable, you can avoid unintended global changes to your table. **sqlwarn4** is set to **W** if float-to-decimal conversion is used.

**sqlwarn5** is set to **W** when your program executes an Informix extension to **ANSI** standard syntax, and the **DBANSIWARN** environment variable is defined.

**sqlwarn6** is not used at present.

**sqlwarn7** is not used at present.

By taking corrective action when **sqlca.sqlcode** is negative, you can recover from the failure of an intended modification of your database. By checking for **sqlca.sqlcode = SQLNOTFOUND**, you can write code to process the results of queries only when rows are returned.



## Dynamic SQL Statements and the *sqlda* Structure

Under the following circumstances, the treatment of dynamically defined SQL statements is more complicated than described in Chapter 1 and requires you to use the *sqlda* structure:

1. In non-parameterized SELECT statements where you know neither the number nor the data types of the members of the *select-list*, nor the list of column names or expressions in the SELECT clause
2. In SELECT statements where you do not know the number or data type of the parameters in the WHERE clause
3. In non-SELECT statements where you do not know the number or data type of the input parameters

If you are not using these kinds of statements, you can skip the rest of this section.

Before looking at the procedure you must follow for each of these statements, consider the *sqlda* structure.

### *The sqlda Structure*

When all of its components are fully defined, the *sqlda* structure points to the beginning of a sequence of *sqlvar\_struct* structures that contain the necessary information for each variable in the set.

---

```

struct sqlvar_struct {
    short sqltype;
    short sqllen;
    char *sqldata;
    short *sqlind;
    char *sqlname;
    char *sqlformat;
    short sqltype;
    short sqlilen;
    char *sqlidata;
};

struct sqlda {
    short sqlid;
    struct sqlvar_struct *sqlvar;
};

```

---

## Explanation

- sqltype** is a short integer corresponding to the data type being transferred. The integer correspondences are defined in **sqltypes.h**.
- sqllen** is a short integer that gives the size in bytes of CHAR type data or the qualifier of a DATETIME or INTERVAL value.
- sqldata** is a pointer to the data.
- sqlind** is a pointer to a short integer indicator variable.
- sqlname** is a pointer to a character array containing the column name or display label being transferred.
- sqlformat** is reserved for future use.
- sqltype** is a short integer indicator variable type. The integer correspondences are defined in **sqltypes.h**.
- sqlilen** is a short integer indicator length in bytes.
- sqlidata** is a pointer to indicator data.

- sqld** is a short integer that is the number of values in the **sqlvar** array.
- sqlvar** is a pointer to an array of **sqlvar\_struct** structures.

## *Non-Parameterized SELECT Statements*

In the SELECT statements described earlier, the values returned from the query are placed into host variables that are listed in an INTO clause. When you create a SELECT statement interactively after you compile your program, you cannot use an INTO clause since the host variables are not directly available. You must use an **sqlda** structure.

There are several steps in this process:

1. Declare a pointer to an **sqlda** structure (name it **udesc**).
2. PREPARE the SELECT statement and give it a statement identifier (let it be **u\_query**).
3. Execute the statement

---

```
$ describe u_query into udesc;
```

---

This statement causes **udesc->sqlvar** to point to a sequence of partially filled **sqlvar\_struct** structures. (**udesc->sqld** gives the number of **sqlvar\_struct** structures.) The components of each **sqlvar\_struct** structure filled by the DESCRIBE statement for each item of the *select-list* are **sqltype**, **sqllen** (for CHAR type data or a qualifier for DATETIME or INTERVAL), and **sqlname**.

4. Allocate memory for the expected values and assign pointers to them (**sqldata** and, optionally, **sqlind**) in each **sqlvar\_struct** structure. This involves your using a function like **malloc()** and assigning proper word boundaries, depending upon the data type and, in the case of CHAR variables, the length of the variable.



5. Declare a cursor for the PREPARED statement identifier (name the cursor `u_cursor`). All dynamically defined SELECT statements must have a declared cursor. For example:

---

```
$ declare u_cursor cursor for u_query;
```

---

6. To open the cursor, execute the statement:

---

```
$ open u_cursor;
```

---

7. Then execute the following statement:

---

```
$ fetch u_cursor using descriptor udesc;
```

---

This statement assigns values to the variables pointed to by the various **sqldata** pointers.

8. Optionally, repeat the FETCH statement.

9. Close the cursor.

To illustrate the process, consider the following program fragment in which all error checking has been suppressed. This example assumes that a SELECT statement has been assembled at run time and is stored in the string `q_string`. Use the following statement:

---

```
select order_num, order_date from orders
```

---

```

#include sqlca;
#include sqllda;
.
.
.
/* declare pointer to structure that
   apportions the data from each row */
struct sqllda *q_desc;

/* declare host variable to hold
   statement string */
$      char q_string[128];

/* make stores the current database */
$      database stores;
.
.
.
/* At this point the SELECT statement
   is assigned to q_string */

/* prepare the SELECT statement */
$      prepare q_id from $q_string;

/* identify the select-list */
$      describe q_id into q_desc;

        /* this section of the code must
           allocate variables to receive
           the data from the rows and
           set the pointers in q_desc.
           See the following discussion */

/* associate q_cursor with SELECT
   statement */
$      declare q_cursor cursor for q_id;

/* open q_cursor; create active set */
$      open q_cursor;

/* loop through rows in active set */
        for(;;)
        {

/* fetch next row from the active set */
$          fetch q_cursor
              using descriptor q_desc;

/* process data if fetch returned a row */
          if (sqlca.sqlcode == 0)
          {
              /* process data */
              .
              .
              .
          }
          else
          {
              /* out of data */
              break;
          }
        }
}

```

After including the header files and declaring the relevant variables, the program selects the **stores** database as the current database. Once the query has been stored in the string **q\_string**, it is

PREPARED and associated with the identifier **q\_id**. Since the statement is a query, the program then DECLAREs a cursor **q\_cursor** so that the rows that are returned can be examined one at a time. The OPEN statement that follows opens the cursor and defines an active set of rows for the SELECT statement.

The complex part of this program fragment follows the DESCRIBE statement and is not shown in detail. The **sqlda** component **q\_desc->sqld** now has the value 2, since two columns were selected from the **orders** table. **q\_desc->sqlvar[0]** is the **sqlvar\_struct** structure that contains information on the **order\_num** column of the **orders** table. **q\_desc->sqlvar[1]** is the **sqlvar\_struct** structure that contains information on the **order\_date** column of the **orders** table. **q\_desc->sqlvar[0].sqltype**, for example, gives the SQL data type of the first column (**order\_num**) requested.

It is your responsibility to allocate memory storage for each of the variables returned by the query. You must set the **sqldata** pointer for each value returned to the location that will receive the value. This process may require aligning data types with proper word boundaries. You can use the **rtypalign** function described in Chapter 5 for this purpose. You can also change the SQL data type to the data type with which you want to work.

When you dynamically allocate **dtime\_t** structures to receive DATETIME values, you can set their qualifiers from the associated **sqllen** fields. Alternatively, you may compose a different qualifier (using the values defined in **datetime.h**) and the database value will be extended to match it.

When you dynamically allocate an **intrvl\_t** structure to receive an INTERVAL value, you should always initialize its qualifier from **sqllen**. Alternatively you may set its qualifier to zero, and the qualifier from the database column will be set during the fetch.

In the preceding illustration, the DESCRIBE statement allocated memory for the necessary number of **sqlvar\_struct** structures. It filled in **sqltype** and **sqlname**, and set to NULL the pointers **sqldata** and **sqlind**. It is your responsibility to set **sqldata** and **sqlind** (if desired) to point to appropriate memory. If you do not use the DESCRIBE statement to create the **sqlvar\_struct** structures, you must remember to set the indicator variable pointers to NULL.

The example program **unload.ec** at the end of this chapter illustrates the use of the **sqlda** structure.



## *Parameterized SELECT Statements*

Parameterized SELECT statements are statements in which input variables are to be supplied at run time. Since DESCRIBE statements examine only the *select-list*—that is, the list of column names or expressions in the SELECT clause—they do not tell you about parameters in the WHERE clauses.

You must know the number of parameters in the SELECT statement and their data types. Unless you are writing a general-purpose, interactive interpreter, you generally have this information. If you do not have it, you must write code that determines not only how many question marks (?) appear in the dynamic query, but also to what data type they belong.

When you know the number of parameters and their data types, you have two options: either declare an equal number of host variables, or allocate memory for an `sqlda` structure that you will use to pass data to the query.

### **Using Host Variables**

If the host variables corresponding to the parameters in the SELECT statement are `hvar1`, `hvar2`, and `hvar3`, you must execute the OPEN statement as shown in the following program fragment:

---

```

$include sqlca
...
/* declare parameter variables */
$      long  hvar1;
$      long  hvar2;
$      long  hvar3;

/* make stores the current database */
$      database stores;

/* prepare the select statement */
$      prepare q_id from
           "select order_num, customer_num
            from orders
            where order_date = ?
                  or paid_date = ?
                  or ship_date = ?";

/* associate q_cursor with SELECT
   statement */
$      declare q_cursor cursor for q_id;

/* among other things, the next section of code
   assigns values to hvar1, hvar2, and hvar3 */

...

/* open q_cursor; create active set */
$      open q_cursor
           using $hvar1, $hvar2, $hvar3;
...

```

---

## Using the *sqlda* Structure

If there are parameters in the SELECT statement and you choose to input values through an *sqlda* structure, you must allocate sufficient memory and supply the following data: *sqld* (= number of parameters), *sqlvar*, and for each parameter *sqltype*, *sqllen* (for CHAR, DATETIME, and INTERVAL) *sqldata*, and *sqlind* (which must be set to NULL). If *in\_vals* is a pointer to the *sqlda* structure that contains the parameter data, you make the query with the following statement:

---

```

$      open q_cursor using descriptor in_vals;

```

---

## *Parameterized Non-SELECT Statements*

The situation for non-SELECT statements that contain parameters is essentially the same as for SELECT statements, except that the EXECUTE rather than the OPEN statement is used to indicate the parameters. If the number of parameters and their data types are known at compile time, an SQL statement that has been PREPARED with the identifier `stateid` can be run with the statement:

---

```
$    execute stateid using $hvar1, $hvar2, $hvar3;
```

---

If the number of parameters is not known until run time and the parameter data is entered into the `sqlda` structure pointed to by `in_vals`, the dynamic statement is run with the statement:

---

```
$    execute stateid using descriptor in_vals;
```

---

## *The ESQL/C Preprocessor*

INFORMIX-ESQL/C statements are processed and converted into C source lines before the C compiler receives them. It is not possible to use the normal C preprocessor to do conditional compilation of INFORMIX-ESQL/C statements, because they have already been processed. Therefore, an INFORMIX-ESQL/C preprocessor has been included in the software to allow conditional compilation of INFORMIX-ESQL/C code.

## *Preprocessor Support*

The INFORMIX-ESQL/C preprocessor allows conditional compilation to be applied to INFORMIX-ESQL/C code. The preprocessor handles these statements:

**\$include** includes a source file in the input at that point.

**\$define** assigns a compile-time value to a name.



**\$undef** removes a **\$defined** name.

**\$ifdef** tests a name and executes subsequent statements if it has been **\$defined**.

**\$ifndef** tests a name and executes subsequent statements if it has not been **\$defined**.

**\$else** begins an alternate section to an **\$ifdef** or **\$ifndef** condition.

**\$endif** closes an **\$ifdef** or **\$ifndef** condition.

You can nest **\$include** statements to a depth of eight. You can write an **\$include** statement with or without quotation marks around the pathname:

---

```
$include pathname;  
$include "pathname";
```

---

When you use the first form, the preprocessor looks for the included file in this sequence:

1. In the current directory
2. In the directory **\$INFORMIXDIR/incl** (where **\$INFORMIXDIR** represents the contents of the environment variable of that name)
3. In the directory **/usr/include**

The remaining statements have the same syntax and effect as their counterparts in the C preprocessor, except that they take effect during input to the INFORMIX-ESQL/C preprocessor.

The **\$define** statement is limited to defining only integer constants. It does not support definition of string constants or parameterized macros.

The preprocessor does not support a generalized **\$if** statement, only the **\$ifdef** and **\$ifndef** statements that test whether a name has been **\$defined**.

## Examples

---

```
$define MAXROWS 25;
$define USETRANSACTIONS;
$ifdef USETRANSACTIONS;
$begin work;
$endif;
```

---

Here, the **BEGIN WORK** statement will only be compiled if the name **USETRANSACTIONS** has been **defined**.

The source file also may contain commands for the C compiler's preprocessor. These commands have no effect on **INFORMIX-ESQL/C** statements but take effect in their usual way when the C compiler receives the source file.

## Compiling ESQL/C Routines

Before you can use the C compiler, you must preprocess your code that has **INFORMIX-ESQL/C** statements. The **INFORMIX-ESQL/C** preprocessor converts the embedded statements to C language code. You can then compile the resulting file with the C compiler to create an object file, which you can then link with the **INFORMIX-ESQL/C** libraries and your own libraries. You can use the **esql** command file that is installed with **INFORMIX-ESQL/C** to perform all these tasks.

To preprocess and compile a C program that contains **INFORMIX-ESQL/C** statements, give its file name the extension **.ec** and enter an **esql** command at your system prompt.

### Syntax

---

```
esql [-e] [-esqlcargs] [-otherargs] \
      [-o outfile] [-ansi] esqlfile.ec [othersrc.c...] \
      [otherobj.o...] [-lyourlib.a...] [-lsyslib...]
```

---

### Explanation

- e** preprocess only, no compilation or linking.
- esqlcargs** **esqlc** arguments. The syntax and explanation of options follows:

---

```
esqlc [-gG] [-nln] [-V] [-ansi] [-t type] [-EDname]
      [-icheck] [-EUname] esqlfile.ec
```

---

- g**            number every line (used by debugger).
- G**            no line number (used by debugger; same as **-nln**).
- nln**          no line number (used by debugger; same as **-G**).
- V**            print preprocessor version information.
- ansi**          check for Informix extensions to ANSI standard syntax.
- t type**       specify the compiler being used.
- EDname**       define a user-supplied name to the preprocessor. The **=val** option lets you assign an initial value to the name, for example:  
**-EDMACNAME=62**.
- icheck**       check for a NULL value returned to a host variable that does not have an indicator variable associated with it, and generate an error if such a case exists.
- EUname**       undefine specified preprocessor name flag.
- otherargs**    other arguments that are passed to the C compiler **cc**.
- o**            next argument is executable program name.
- ansi**          check for Informix extensions to ANSI standard syntax.
- esqlfile.ec**   your C program.
- othersrc.c**    other C source files you want to compile and link.
- otherobj.o**    other C object files you want to link.



**-lyourlib.a** your own special libraries that you want to link.

**-lsyslib** other system libraries that you want to link.

## Notes

1. There are two ways to check for Informix extensions to ANSI standard syntax when you compile an **INFORMIX-ESQL/C** program.
  - You can set the **DBANSIWARN** environment variable. (See Appendix C.) Thereafter, every time you compile or run a program, the program is checked automatically for ANSI compatibility.
  - If **DBANSIWARN** is not defined, you can include the **-ansi** command line option whenever you want to check a program for ANSI compatibility at compile time. This is not a valid option at run time.
2. The **-ansi** option asks the compiler to verify that all SQL statements meet ANSI standards. If you use the **-ansi** flag to compile a program that contains Informix extensions to ANSI standard syntax, or if the **DBANSIWARN** environment variable is defined, **esql** generates warning messages on the screen during the compile. (At run time, the **DBANSIWARN** environment variable causes **sqlwarn5** to be set to **W** when a non-ANSI statement is executed.)
3. The **-icheck** option tells the preprocessor to generate an error when an SQL statement returns a **NULL** value to a host variable that does not have an indicator variable associated with that host variable. If you do not compile using the **-icheck** flag, no error is generated in this situation. (See the earlier discussion of indicator variables.)
4. The **INFORMIX-ESQL/C** preprocessor treats uppercase and lowercase letters as distinct identifiers.

In the compiler syntax shown earlier, **outfile** is the name of the executable output file and **esqlfile.ec** is your C program. **esql** first preprocesses the embedded SQL statements in **esqlfile.ec** and, if successful, produces a file **esqlfile.c**. The **-e** option allows you to request just the preprocessor step with no compilation or linking.

If you have properly set the environment variable **INFORMIXDIR** (see Appendix C), **esql** passes all other arguments (**-otherargs**), **esqlfile.c**, and other C source files (**othersrc.c**) straight through to the C compiler **cc** to produce **esqlfile.o** and **othersrc.o**. These files are then linked with the appropriate **INFORMIX-ESQL/C** library routines, along with other C object files (**otherobj.o**) that you include on the command line. You must explicitly include other system libraries on the command line (for example, **libm.a** for mathematical functions).

Whether or not the **-ansi** parameter is used, compiled user programs can make run-time checks on Informix extensions to ANSI SQL syntax by using the **sqlca** (SQL Control Area) structure, if the **DBANSIWARN** environment variable is set. (See the section “Error Handling and the **sqlca** Structure” for details.) Flags are set to indicate:

- After opening a database, whether it uses transactions
- After opening a database, whether it operates as **MODE ANSI**
- After opening a database, whether the engine is **INFORMIX-OnLine**
- Whenever an Informix extension to SQL is executed

## Examples

The four example programs in this section become increasingly more complicated.

Each of the first three examples displays on the screen the names of all the customers in the **stores** table whose last names begin with **C** through **Z**.

- The first uses a **SELECT** statement with no free parameters.
- The second uses a **PREPARE** statement on a query with an unknown parameter in the **WHERE** clause.
- The third is similar to the second, but uses the **DESCRIBE** statement to fill an **sqlda** structure rather than using host variables.

The fourth example demonstrates many of the features of **INFORMIX-ESQL/C** by providing the code to analyze an arbitrary **SELECT** statement and to create an unloaded system file containing the data from the resulting table with a bar ( **|** ) delimiter between the fields.

## *demo1.ec*

```
#include <stdio.h>
#include sqlca;

/* Uncomment the following line if the database has
   transactions:      */

/* $define TRANS;    */

#define FNAME_LEN    15;
#define LNAME_LEN    15;

main( )

{
    $char fname[ FNAME_LEN + 1 ];
    $char lname[ LNAME_LEN + 1 ];

    printf( "DEMO1 Sample ESQL program running.\n\n");

    $database stores;

    $declare democursor cursor for
        select fname, lname
            into $fname, $lname
            from customer
            where lname > "C";

    $ifdef TRANS;
    $begin work;
    $endif;

    $open democursor;

    for ( ;; )
    {
        $fetch democursor;
        if (sqlca.sqlcode != 0) break;
        printf("%s %s\n",fname, lname);
    }

    $close democursor;

    $ifdef TRANS;
    $commit work;
    $endif;

    printf("\nProgram Over.\n");
}
```



## *demo2.ec*

---

```
#include <stdio.h>
#include sqlca;

/* Uncomment the following line if the database has
   transactions: */

/* $define TRANS; */

#define FNAME_LEN 15;
#define LNAME_LEN 15;

main( )
{
    $char demoquery[80];
    $char queryvalue[2];
    $char fname[ FNAME_LEN + 1 ];
    $char lname[ LNAME_LEN + 1 ];

    /* These next four lines have hard-wired both
       * the query and the value for the parameter.
       * This information could have been entered
       * from the terminal and placed into the strings
       * demoquery and queryvalue, respectively.
       */
    sprintf(demoquery, "%s %s",
            "select fname, lname from customer",
            "where lname > ? ");
    sprintf(queryvalue, "C");

    printf("DEMO2 Sample ESQL program running.\n\n");

    $database stores;
    $prepare qid from $demoquery;
    $declare democursor cursor for qid;

    $ifdef TRANS;
    $begin work;
    $endif;
```

```

$open democursor using $queryvalue;

if (sqlca.sqlcode)
    printf("%s %d\n",
           "sqlca.code, after the cursor open, is",
           sqlca.sqlcode);
for (;;)
{
    $fetch democursor into $fname, $lname;
    if (sqlca.sqlcode != 0) break;
    printf("%s %s\n", fname, lname);
}
if (sqlca.sqlcode != SQLNOTFOUND)
    printf("%s %d\n",
           "sqlca.code, after fetch, is",
           sqlca.sqlcode);

$close democursor;

#ifdef TRANS;
$commit work;
#endif;

printf("\nProgram Over.\n");
}

```

---

## *demo3.ec*

---

```
#include <stdio.h>
#include sqlca;
#include sqllda;

/* Uncomment the following line if the database has
   transactions: */

/* $define TRANS; */

#define FNAME_LEN 15;
#define LNAME_LEN 15;

main( )
{
    struct sqllda *demodesc;
    $char demoquery[80];
    $char queryvalue[2];
    $char fname[ FNAME_LEN + 1 ];
    $char lname[ LNAME_LEN + 1 ];

    /* These next four lines have hard-wired both
       * the query and the value for the parameter.
       * This information could have been entered
       * from the terminal and placed into the strings
       * demoquery and queryvalue, respectively.
       */
    sprintf(demoquery, "%s %s",
            "select fname, lname from customer",
            "where lname > ? ");
    sprintf(queryvalue, "C");

    printf("DEMO3 Sample ESQL program running.\n\n");
    $database stores;
    $prepare qid from $demoquery;
    $declare democursor cursor for qid;

    $ifdef TRANS;
    $begin work;
    $endif;

    $open democursor using $queryvalue;
    $describe qid into demodesc;
```



```

/* Print out what DESCRIBE returns */
prsqlda(demodesc->sqlvar);
prsqlda(demodesc->sqlvar+1);

/* Assign the data pointers */
demodesc->sqlvar[0].sqldata = fname;
demodesc->sqlvar[1].sqldata = lname;

/* Null the indicator pointers */
demodesc->sqlvar[0].sqlind = NULL;
demodesc->sqlvar[1].sqlind = NULL;

/* Allow for the trailing null character in C
   character arrays */
demodesc->sqlvar[0].sqlllen += 1;
demodesc->sqlvar[1].sqlllen += 1;

for (;;)
{
    $fetch democursor using descriptor demodesc;
    printf("sqlca.code, after fetch, is %d\n",
           sqlca.sqlcode);
    if (sqlca.sqlcode != 0) break;
    printf("%s %s\n", fname, lname);
}

$close democursor;

#ifdef TRANS;
$commit work;
#endif;

printf("\nProgram Over.\n");
}

prsqlda(sp)
    register struct sqlvar_struct *sp;
{
    printf("type = %d\n", sp->sqltype);
    printf("len = %d\n", sp->sqlllen);
    printf("data = %lx\n", sp->sqldata);
    printf("ind = %lx\n", sp->sqlind);
    printf("name = %s\n", sp->sqlname);
}

```

## *unload.ec*

The following code contains a minimal **main( )** routine that makes stores the current database and calls the function **unload( )**.

The function **unload( )** is defined next. It takes two arguments: a string that contains a **SELECT** statement and a string that contains the name of a file. The structure of the code follows the pattern outlined earlier in this chapter in the section "Non-Parameterized **SELECT** Statements." The **SELECT** statement passed to **unload( )** is **PREPARED** and, if successful, is **DESCRIBEd** into the **sqlda** structure pointed to by **udesc**.

If the **DESCRIBE** was successful, the output file is opened and three major steps are performed. First, **\*udesc** is analyzed to determine the type and memory requirements of each item in the *select-list* of the **SELECT** statement. Second, memory (**buffer**) is allocated to receive a row of the table returned by the **SELECT** statement. Third, pointers (**col->sqldata**) are set in the allocated memory to receive each item in the *select-list*.

A cursor is **DECLAREd** for the **SELECT** statement and the cursor is **OPENEd**. The next section of code loops through a series of **FETCHes** until there are no more rows in the active set (**sqlca.sqlcode != 0**). In the loop, each item in a row is retrieved and written out to the output file with a delimiter ( **|** ) between each item. A count is kept of the number of rows.

At the end, if the process was successful, a message is written to standard output, giving the number of rows unloaded.

```

#include <stdio.h>
#include sqlca;
#include sqlda;
#include sqltypes;

#define FLDSIZE 40

/* This program handles ONLY the following data types:
 *
 *      SMALLFLOAT      FLOAT      MONEY
 *      DECIMAL         CHAR
 *
 * It does not handle any other SQL data type, nor does it
 * handle nulls. (Adding such support is easy.)
 */

int has_trans; /* True if DB has transactions. */

main()
{
    /* Use the status code returned from the DATABASE
     * statement to check if the database has transactions.
     */

    $ database stores;
    has_trans = ( sqlca.sqlwarn.sqlwarn1 == 'W' );
    unload("select lname, company from customer",
           "cust.unl");
}

unload (slctstmt, fname)
$char *slctstmt;
char *fname;
{
    register int pos;
    register char *cp;
    register int len;
    register int i;
    char field[FLDSIZE+1];
    char *fieldp;
    struct sqlda *udesc;
    struct sqlvar_struct *col;
    FILE *unlfile;
    char delim = '|';
    char *buffer = NULL;
    char *malloc();
    char *rtypalign();

```



```

$   prepare usqlobj from $slctstmt;
    if ( sqlca.sqlcode != 0 )
        return;
$   describe usqlobj into udesc;
    if ( sqlca.sqlcode != 0 )
        return;
    unfile = fopen(fname,"w");
    if ( unfile == NULL )
    {
        fprintf(stderr,"Cannot open file '%s'\n", fname);
        return ;
    }

    /* Step 1: analyze udesc to determine type and memory requirements
     *          of each item in the select list.  rtypalign() returns a
     *          pointer to the next appropriate boundary (starting at
     *          pos) for the indicated data type.
     */
    pos = 0;
    for (col = udesc->sqlvar, i = 0; i < udesc->sqld; col++, i++)
    {
        switch(col->sqltype)
        {
            case SQLSMFLOAT:
                col->sqltype = CFLOATTYPE;
                break;

            case SQLFLOAT:
                col->sqltype = CDOUBLETYPE;
                break;

            case SQLMONEY:
            case SQLDECIMAL:
                col->sqltype = CDECIMALTYPE;
                break;

            case SQLCHAR:
                col->sqltype = CCHARTYPE;
                break;

            default:
                /* The program does not handle INTEGER,
                 * SMALL INTEGER, DATE, SERIAL or other
                 * data types.  Do nothing if we see
                 * an unsupported type. */
                return;
        }
    }
    col->sqllen = rtypmsize(col->sqltype,col->sqllen);
    pos = (int) rtypalign(pos,col->sqltype) + col->sqllen;
    col->sqlind = NULL;
}

    /* Step 2: Allocate memory to receive a row of the table returned
     *          by the SELECT statement.  The variable pos has an integer
     *          value equal to the number of bytes occupied by the row.
     */
    buffer = malloc(pos);
    if ( buffer == NULL )
    {
        fprintf(stderr,"Out of memory\n");
    }

```

```

    return;
}

/* Step 3: Set pointers in the allocated memory to receive each
 *          item in the select list.
 */
cp = buffer;
for (col = udesc->sqlvar, i = 0; i < udesc->sqld; col++, i++)
{
    cp = rtypalign(cp,col->sqltype);
    col->sqldata = cp;
    cp += col->sqllen;
}

/* Step 4: Fetch each row of the query, convert to ASCII format, and
 *          write to the output file.
 */
$ declare usqlcurs cursor for usqlobj;

if ( has_trans )
$     begin work;

$     open usqlcurs;
$     fetch usqlcurs using descriptor udesc;
while (sqlca.sqlcode == 0 )
{
    for (col=udesc->sqlvar, i = 0; i < udesc->sqld; col++, i++)
    {
        byfill(field,FLDSIZE+1,0);
        fieldp = field;
        switch (col->sqltype)
        {

            case CFLOATTYPE:
                sprintf(field,"%f",
                    (double) *((float *) (col->sqldata)));
                break;

            case CDOUBLETYPE:
                sprintf(field,"%f",*((double *) (col->sqldata)));
                break;

            case CDECIMALTYPE:
                dectoasc(col->sqldata,field,FLDSIZE,-1);
                break;

            case CCHARTYPE:
                fieldp = col->sqldata;
                break;

            default:
                /* Unsupported data type. */
                break;
        }
    }
}

```

```

    }
    len = stleng(fieldp);
    cp = fieldp + len - 1;
    while ( len > 1 && *cp == ' ' )
    {
        *cp = '\0';
        len--, cp--;
    }
    fputs(fieldp,unlfile);
    putc(delim,unlfile);
    }
    putc('\n',unlfile);
$   fetch usqlcurs using descriptor udesc;
}

fclose(unlfile);
free(buffer);
if ( sqlca.sqlcode && sqlca.sqlcode != SQLNOTFOUND )
{
    /* display ERROR message*/
    fprintf(stderr,"unload failed sqlcode = %ld\n", sqlca.sqlcode);
}
else
{
    fprintf(stderr,"unloaded %ld rows\n" , sqlca.sqlerrd[2]);
}

$   close usqlcurs;
    if ( has_trans )
$   commit work;
}

```

---



# **Chapter 3**

## **SQL Statements**



## Chapter 3 Table of Contents

SQL Statements .....	5
Statements Supported Only on INFORMIX-SE .....	6
Statements Supporting INFORMIX-OnLine Enhancements .....	7
INFORMIX-ESQL/C Extensions to ANSI Syntax .....	7
SELECT Statement.....	8
DECLARE Statement.....	9
UPDATE Statement.....	9
GRANT Statement.....	10
CREATE TABLE Statement.....	10
CREATE VIEW Statement.....	11
Definition of Statements.....	11
ALTER INDEX.....	12
ALTER TABLE (O) .....	14
BEGIN WORK .....	18
CLOSE .....	20
CLOSE DATABASE.....	22
COMMIT WORK.....	23
CREATE AUDIT .....	24
CREATE DATABASE (O).....	26
CREATE INDEX.....	29
CREATE SYNONYM .....	32
CREATE TABLE (O).....	34
CREATE VIEW .....	43
DATABASE .....	46
DECLARE.....	48
DELETE .....	53
DESCRIBE.....	56
DROP AUDIT .....	59
DROP DATABASE.....	60
DROP INDEX .....	62
DROP SYNONYM.....	63
DROP TABLE .....	65
DROP VIEW .....	67
EXECUTE.....	68
EXECUTE IMMEDIATE .....	70
FETCH.....	72
FLUSH.....	75
FREE .....	77
GRANT .....	79
INSERT.....	83
LOCK TABLE .....	87



OPEN .....	89
PREPARE .....	92
PUT .....	96
RECOVER TABLE.....	99
RENAME COLUMN.....	101
RENAME TABLE .....	103
REVOKE.....	105
ROLLBACK WORK.....	108
ROLLFORWARD DATABASE.....	109
SELECT.....	110
SET EXPLAIN .....	111
SET LOCK MODE (O) .....	114
START DATABASE.....	116
UNLOCK TABLE.....	119
UPDATE .....	120
UPDATE STATISTICS.....	124
WHENEVER.....	126
 The SELECT Statement.....	 129
SELECT Clause.....	134
INTO Clause .....	137
FROM Clause.....	140
WHERE Clause.....	142
Comparison Condition.....	142
Join Condition .....	152
Condition with a Subquery.....	155
GROUP BY Clause .....	159
HAVING Clause.....	161
ORDER BY Clause.....	163
INTO TEMP Clause.....	165
UNION Operator .....	167
Functions in SQL Statements.....	170
Aggregate Functions .....	171
LENGTH( ) .....	173
DATE( ) .....	174
DAY( ) .....	175
MDY( ) .....	176
MONTH( ).....	177
WEEKDAY( ) .....	178
YEAR( ).....	179
CURRENT .....	180
EXTEND( ).....	182

# SQL Statements

This chapter describes those SQL statements that can be embedded in a C program. Use it along with Chapter 2, where you find definitions of C structures, host variables, and indicator variables, as well as instructions on how to prepare a C program for the **INFORMIX-ESQL/C** preprocessor and how to compile it.

The SQL statements listed below are described in detail, in alphabetical order, throughout this chapter. Because the **SELECT** statement and its clauses (**FROM**, **INTO**, **WHERE**, **GROUP BY**, **HAVING**, **ORDER BY**, and **INTO TEMP**) require special attention, they are described in a later section of this chapter.

The SQL statements and statement options that apply only to dynamically defined statements are marked with a cross symbol (†). Since you need dynamically defined statements only in advanced applications, you can ignore these statements and options for most purposes.

---

ALTER INDEX	FETCH
ALTER TABLE	FLUSH
BEGIN WORK	FREE†
CLOSE	GRANT
CLOSE DATABASE	INSERT
COMMIT WORK	LOCK TABLE
CREATE AUDIT	OPEN
CREATE DATABASE	PREPARE†
CREATE INDEX	PUT
CREATE SYNONYM	RECOVER TABLE
CREATE TABLE	RENAME COLUMN
CREATE VIEW	RENAME TABLE
DATABASE	REVOKE
DECLARE	ROLLBACK WORK
DELETE	ROLLFORWARD DATABASE
DESCRIBE†	SELECT
DROP AUDIT	SET EXPLAIN
DROP DATABASE	SET LOCK MODE
DROP INDEX	START DATABASE
DROP SYNONYM	UNLOCK TABLE
DROP TABLE	UPDATE
DROP VIEW	UPDATE STATISTICS
EXECUTE†	WHENEVER
EXECUTE IMMEDIATE†	

---

## ***Statements Supported Only on INFORMIX-SE***

The following SQL statements are supported only on **INFORMIX-SE**. They cannot be used in **INFORMIX-OnLine** applications.

---

CREATE AUDIT  
 DROP AUDIT  
 RECOVER TABLE  
 ROLLFORWARD DATABASE  
 START DATABASE

---



## ***Statements Supporting INFORMIX-OnLine Enhancements***

The INFORMIX-OnLine database engine recognizes extensions to several SQL statements. Refer to the *INFORMIX-OnLine Programmer's Manual* for details of the additional functionality available with INFORMIX-OnLine.

INFORMIX-ESQL/C statements that support INFORMIX-OnLine enhancements are listed below. They are identified in the descriptions that follow with an (O) next to the name of the statement. ***Do not include the O when you type the SQL statement.***

---

ALTER TABLE (O)  
CREATE DATABASE (O)  
CREATE TABLE (O)  
SET LOCK MODE (O)

---

## ***INFORMIX-ESQL/C Extensions to ANSI Syntax***

You can check your INFORMIX-ESQL/C programs for ANSI compatibility by defining the DBANSIWARN environment variable before you compile or run a program, or by using the **-ansi** flag as a command line parameter when you compile. See "Setting Environment Variables" in the Preface or Appendix C for information on how to set this environment variable.

If you have not defined DBANSIWARN and you want to check a program for ANSI compatibility at compile time, use the **-ansi** flag. For example:

---

```
esql -ansi file.ec -o progname
```

---

At compile time, regardless of the type of database, the **-ansi** flag or DBANSIWARN environment variable causes warning messages to be displayed to the screen whenever a non-ANSI statement is encountered in your program.



At run time, if you have set the DBANSIWARN environment variable, non-ANSI statements cause **sqlwarn0** and **sqlwarn5** to be set to **W**. (See Chapter 2 for more information about the **sqlca.sqlwarn** character string.)

The following statements are extensions to ANSI syntax and generate warnings when Informix extension checking has been initiated with the DBANSIWARN environment variable or the **-ansi** flag:

---

ALTER INDEX	DROP VIEW
ALTER TABLE	FLUSH
BEGIN WORK	FREE
CLOSE DATABASE	GRANT
CREATE AUDIT	LOCK TABLE
CREATE DATABASE	PUT
CREATE INDEX	RECOVER TABLE
CREATE SYNONYM	RENAME COLUMN
CREATE TABLE	RENAME TABLE
CREATE VIEW	REVOKE
DATABASE	ROLLFORWARD DATABASE
DROP AUDIT	SET EXPLAIN
DROP DATABASE	SET LOCK MODE
DROP INDEX	START DATABASE
DROP SYNONYM	UNLOCK TABLE
DROP TABLE	UPDATE STATISTICS

---

**Note:** The **BEGIN WORK** statement generates a warning at compile time only.

The following sections list keywords or features that are extensions to ANSI standard syntax. A warning is generated if you include these keywords or features in a program and you have initiated Informix extension checking with the DBANSIWARN environment variable or the **-ansi** flag.

## SELECT Statement

The following keywords or features are Informix extensions to the **SELECT** statement:

- **UNIQUE** keyword
- **OUTER** keyword

- Column labels
- MATCHES keyword
- UNITS keyword
- Numbers as position indicators (for example, in the GROUP BY statement)
- INTO TEMP clause
- Column subscripts
- The following functions:
  - LENGTH()
  - DATE()
  - DAY()
  - MDY()
  - MONTH()
  - WEEKDAY()
  - YEAR()
  - CURRENT
  - EXTEND()
  - TODAY

## **DECLARE Statement**

The following keywords are Informix extensions to the DECLARE statement:

- WITH HOLD
- SCROLL
- INSERT
- FOR UPDATE

## **UPDATE Statement**

Specifying multiple columns in an UPDATE statement is an extension to the ANSI standard.

## GRANT Statement

The following keywords are Informix extensions to the GRANT statement:

- CONNECT
- RESOURCE
- DBA
- AS

Use of the GRANT statement in INFORMIX-ESQL/C generates a run-time warning when Informix extension checking is initiated. The ANSI standard requires that the GRANT statement be issued within the CREATE SCHEMA AUTHORIZATION statement. For more information about the CREATE SCHEMA AUTHORIZATION statement, refer to the *INFORMIX-SQL Reference Manual*.

## CREATE TABLE Statement

The following features and keywords are Informix extensions to the CREATE TABLE statement:

- TEMP keyword
- DISTINCT keyword
- UNIQUE CONSTRAINT keywords
- IN keyword
- The following data types:
  - SERIAL
  - DATE
  - MONEY
  - SMALLFLOAT
  - DATETIME
  - INTERVAL

Use of the CREATE TABLE statement in INFORMIX-ESQL/C generates a run-time warning when Informix extension checking is initiated. The ANSI standard requires that the CREATE TABLE statement be issued within the CREATE SCHEMA AUTHORIZATION statement. For more information about the CREATE SCHEMA

AUTHORIZATION statement, refer to the *INFORMIX-SQL Reference Manual*.

## CREATE VIEW Statement

Use of the CREATE VIEW statement in INFORMIX-ESQL/C generates a run-time warning when Informix extension checking is initiated. The ANSI standard requires that the CREATE VIEW statement be issued within the CREATE SCHEMA AUTHORIZATION statement. For more information about the CREATE SCHEMA AUTHORIZATION statement, refer to the *INFORMIX-SQL Reference Manual*.

---

**Note:** INFORMIX-OnLine supports additional keywords that are extensions. See the *INFORMIX-OnLine Programmer's Manual* for more information.

---

## Definition of Statements

The following section describes the INFORMIX-ESQL/C statements. The statements appear in alphabetical order.



# ALTER INDEX

## Overview

Use the ALTER INDEX statement to cluster a table in the order of an existing index or to release an index from the cluster attribute.

## Syntax

---

ALTER INDEX *index-name* TO [NOT] CLUSTER

---

## Explanation

ALTER INDEX      are required keywords.

*index-name*      is the SQL identifier of an existing index.

TO                is a required keyword.

NOT              is an optional keyword.

CLUSTER        is a required keyword.

## Notes

1. The TO CLUSTER option causes INFORMIX-ESQL/C to reorder the rows in the physical table to agree with the order of *index-name*. Reordering causes the entire file to be rewritten and requires sufficient disk space to maintain two copies of the table. The process may take a long time.
2. Since there can be only one clustered index per table, you must use the NOT option to release the cluster attribute from one index before assigning it to another. The NOT option does not affect the physical table; it merely drops the cluster attribute on *index-name* from the system catalogs.

3. When INFORMIX-ESQL/C executes ALTER INDEX with the TO CLUSTER option, it locks the table IN EXCLUSIVE MODE. If some other process is using the table to which *index-name* belongs, INFORMIX-ESQL/C cannot execute ALTER INDEX with the TO CLUSTER option and returns an error.
4. The ALTER INDEX statement cannot be rolled back.
5. Over time you can expect the benefit of an earlier cluster to disappear. You can recluster the table by issuing another ALTER INDEX TO CLUSTER statement on the clustered index.
6. You do not need to drop a cluster index before issuing another ALTER INDEX TO CLUSTER statement on a currently clustered index.

### Example

The following example creates two indexes on the **orders** table and clusters the physical table in ascending order on the **customer\_num** column. Later, the example clusters the physical table in ascending order on the **order\_num** column.

---

```
$ create unique index ix_ord
    on orders (order_num);

$ create cluster index ix_cust
    on orders (customer_num);
...

$ alter index ix_cust to not cluster;

$ alter index ix_ord to cluster;
```

---

### Related Statement

CREATE INDEX

# ALTER TABLE (O)

## Overview

Use the ALTER TABLE statement to add a column to a table, delete a column from a table, modify the data type of a column, add a UNIQUE CONSTRAINT to a column or composite list of columns, or drop a UNIQUE CONSTRAINT associated with a column or composite list of columns.

## Syntax

---

ALTER TABLE *table-name*

```
{ADD (newcol-name newcol-type [NOT NULL]
  [UNIQUE [CONSTRAINT constr-name]] [, ...]) [BEFORE oldcol-name]
| DROP (oldcol-name [, ...])
| MODIFY (oldcol-name newcol-type [NOT NULL] [, ...])
| ADD CONSTRAINT UNIQUE (oldcol-name [, ...])
  [CONSTRAINT constr-name]
| DROP CONSTRAINT (constr-name [, ...])} [, ...]
```

---

## Explanation

ALTER TABLE	are required keywords.
<i>table-name</i>	is the name of an existing table.
ADD	is a keyword you use to add a column to <i>table-name</i> .
<i>newcol-name</i>	is the name of the column you want to add.
<i>newcol-type</i>	is either the data type of the column you are adding or the data type of the column you are modifying.
NOT NULL	are optional keywords.
UNIQUE	is a keyword specifying that the column or composite column list accepts only unique values.



<b>CONSTRAINT</b>	is a keyword you use to indicate that <i>constr-name</i> is assigned in the statement.
<i>constr-name</i>	is the name of the constraint.
<b>BEFORE</b>	is an optional keyword you use to indicate where you want <i>newcol-name</i> placed in the list of columns. The default is at the end of the list of columns.
<i>oldcol-name</i>	is the name of an existing column.
<b>DROP</b>	is a keyword you use to drop a column from <i>table-name</i> .
<b>MODIFY</b>	is a keyword you use to change the data type of an existing column.
<b>ADD CONSTRAINT</b>	are keywords you use to place a constraint on a column or composite column list.
<b>DROP CONSTRAINT</b>	are keywords you use to drop a <b>UNIQUE CONSTRAINT</b> on a table column.

## Notes

1. In a database created as **MODE ANSI**, the name of a table is qualified by the owner of the table (*owner.table-name*). You must specify *owner* when you refer to a table owned by another user.  
  
The use of the prefix *owner.* is optional in a non-**MODE ANSI** database. **INFORMIX-ESQL/C** does check the accuracy of *owner* if you include it in a statement, however. See the section "Owner Naming" in Chapter 1 for more information.
2. You can use one or more of the **ADD**, **DROP**, **MODIFY**, **ADD CONSTRAINT**, or **DROP CONSTRAINT** clauses, and you can place them in any order. Use a comma to separate the clauses. The actions are performed in the order specified. If any of the actions fails, the entire operation is canceled.
3. You cannot add a **SERIAL** column to a table. You must create a **SERIAL** column with the **CREATE TABLE** statement; you cannot add it with the **ALTER TABLE** statement.



4. You can modify an existing column that formerly permitted NULLs to be NOT NULL, provided that it does not already contain any NULL values. Specify MODIFY with the same *oldcol-name* and data type and the NOT NULL keywords.
5. You can modify an existing column that did *not* permit NULLs to permit NULLs. Specify MODIFY with the *oldcol-name* and the existing data type and omit NOT NULL.
6. When you add a new column to an existing table, it is filled with NULL values. Therefore, you cannot define a new column as NOT NULL, or specify a UNIQUE CONSTRAINT when you add a column, unless the table contains no data.
7. When an existing column is modified to a different data type, all data is converted to the new data type, including numeric to character and character to numeric (assuming the character represents a number). However, if there is a UNIQUE CONSTRAINT, the conversion will take place only if it does not violate the constraint.

If a data conversion would result in the creation of duplicate values (for example by changing a FLOAT to a SMALLFLOAT or reducing the size of a CHAR column), then the ALTER TABLE command will fail, and you will receive error 212, "Cannot add index", and ISAM error 100, "There is already a record with the same value in a unique index".

8. When you drop a column that is part of a multiple-column constraint, you automatically drop the corresponding UNIQUE CONSTRAINT.
9. You must own *table-name*, have DBA privilege, or be granted ALTER permission to use ALTER TABLE.
10. Altering a table on which a view depends may invalidate the view.
11. You cannot use a ROLLBACK WORK statement to undo an ALTER TABLE statement.
12. You can use ALTER TABLE with the ADD CONSTRAINT clause to place a UNIQUE CONSTRAINT on a table column or composite column list. You can use ALTER TABLE with the DROP CONSTRAINT clause to drop a UNIQUE CONSTRAINT.

The following rules apply when adding a **UNIQUE CONSTRAINT** to a column or composite column list:

- The columns can contain only unique values.
  - A **UNIQUE CONSTRAINT** cannot already apply to the columns.
  - An ascending index cannot already apply to the columns.
13. To drop an existing constraint, specify **DROP CONSTRAINT** with the name of the constraint. If a name was not given to the constraint when it was created, the system generated the name. You can query the **sysconstraints** system catalog for the names of constraints.
14. If you own the table or have **ALTER** permission on the table, you can create a constraint on the table and specify yourself as the owner. If you have **DBA** permission, you can create constraints for other users.

## Examples

---

```
$  alter table items
    add (item_weight decimal(6,2)
        before total_price);

$  alter table items
    drop (total_price);

$  alter table items
    modify (manu_code char(4));

$  alter table manufact add constraint unique (manu_name)
    constraint con_name;

$  alter table manufact
    drop constraint (con_name);
```

---

## Related Statements

**RENAME COLUMN, RENAME TABLE, CREATE TABLE**

# BEGIN WORK

## Overview

Use the **BEGIN WORK** statement to start a transaction (a sequence of database operations that are terminated by the **COMMIT WORK** or **ROLLBACK WORK** statement).

## Syntax

---

**BEGIN WORK**

---

## Explanation

**BEGIN WORK**     are required keywords.

## Notes

1. Each row affected by an **UPDATE**, **DELETE**, or **INSERT** statement during a transaction is locked and remains locked throughout the transaction. A transaction that contains a large number of such statements, or that contains statements affecting a large number of rows, may exceed the limits placed by your operating system on the maximum number of simultaneous locks. If you encounter this error, you may need to lock the entire table immediately after beginning the transaction. See the “Locking” section in Chapter 1 for a detailed description of table-level and row-level locking in **INFORMIX-ESQL/C**.
2. Do not use the **BEGIN WORK** statement with a database **CREATED** or **STARTED** as **MODE ANSI**. In a program that accesses a **MODE ANSI** database, the **BEGIN WORK** statement generates a run-time error unless it appears immediately after one of the following statements:

---

CREATE DATABASE  
DATABASE  
START DATABASE

COMMIT WORK  
ROLLBACK WORK

---

The BEGIN WORK statement is not needed in a MODE ANSI database, since transactions are implicit.

### **Related Statements**

COMMIT WORK, ROLLBACK WORK



# CLOSE

## Overview

Use the CLOSE statement when you no longer need to refer to the active set of a SELECT cursor or when you want to flush the insert buffer and close an INSERT cursor.

## Syntax

---

CLOSE *cursor-name*

---

## Explanation

CLOSE is a required keyword.

*cursor-name* is the name of a cursor that has been DECLARED for a SELECT or an INSERT statement.

## Notes

1. If *cursor-name* is associated with a SELECT statement, the CLOSE statement puts the cursor in a closed state and leaves the active set undefined.
2. After you CLOSE a SELECT cursor, you cannot execute a FETCH statement until you reOPEN it.
3. If *cursor-name* is associated with an INSERT statement, the CLOSE statement flushes any rows in the buffer into the database and closes the cursor.
4. After you CLOSE an INSERT cursor, you cannot execute a PUT or FLUSH statement until you reOPEN the cursor.
5. The global variables `sqlca.sqlcode` and `sqlca.sqlerrd[2]` indicate the result of each FLUSH and CLOSE statement for an INSERT cursor.

- If **INFORMIX-ESQL/C** successfully inserts the buffered rows into the database, it sets **sqlca.sqlcode** to zero and **sqlca.sqlerrd[2]** to the number of rows inserted into the database.
- If **INFORMIX-ESQL/C** encounters an error while inserting the buffered rows into the database, it sets **sqlca.sqlcode** to a negative number (specifically, the number of the error message) and sets **sqlca.sqlerrd[2]** to the number of rows successfully inserted into the database.

Buffered rows following the last successfully inserted row are discarded. In this case, **CLOSE** does not close the cursor.

6. Although the **COMMIT WORK** and **ROLLBACK WORK** statements close all open cursors (except cursors **WITH HOLD**), do not use them for this purpose. Explicitly **CLOSE** each **INSERT** cursor before committing the work, so you can check that the insertion was successful.
7. **INFORMIX-ESQL/C** does not provide a global variable containing the total number of rows successfully inserted into the database with an insert cursor. If you want to know the total number of inserts performed, you must set a counter in your program and increment it upon each **PUT** statement.

## Examples

---

```
$ close q_curs;
$ close i_curs;
```

---

## Related Statements

**DECLARE, FETCH, FLUSH, OPEN, PUT**

# CLOSE DATABASE

## Overview

Use the CLOSE DATABASE statement to close the current database.

## Syntax

---

CLOSE DATABASE

---

## Explanation

CLOSE DATABASE      are required keywords.

## Notes

1. Following the CLOSE DATABASE statement, the only legal SQL statements are CREATE DATABASE, DATABASE, and DROP DATABASE, (and, with INFORMIX-SE only, ROLLFORWARD DATABASE and START DATABASE).
2. Issue the CLOSE DATABASE statement before you DROP the current database.
3. The CLOSE DATABASE statement cannot appear in a multi-statement PREPARE.

## Related Statements

CREATE DATABASE, DROP DATABASE

# COMMIT WORK

## Overview

Use the COMMIT WORK statement to commit all modifications made to the database since the beginning of the transaction.

## Syntax

---

COMMIT WORK

---

## Explanation

COMMIT WORK      are required keywords.

## Notes

1. Use the COMMIT WORK statement when you are satisfied that the changes made to the database since the beginning of the transaction are what you want.
2. The COMMIT WORK statement closes all open cursors except those declared as WITH HOLD.
3. See the section "Transactions" in Chapter 1 for details.

## Related Statements

BEGIN WORK, ROLLBACK WORK



# CREATE AUDIT

## Overview

Use the CREATE AUDIT statement to create an audit trail file and to start writing the audit trail.

## Syntax

---

```
CREATE AUDIT FOR table-name IN "pathname"
```

---

## Explanation

CREATE AUDIT  
FOR

are required keywords.

*table-name*

is the name of the table for which you want to create an audit trail file.

IN

is a required keyword.

*pathname*

is the full pathname for the audit trail file. It must be enclosed in quotation marks.

## Notes

1. You create audit trails to keep a record of all modifications of a table. An audit trail is a complete history of all additions, deletions, and updates to the table. **INFORMIX-ESQL/C** can use the audit trail to reconstruct the table from a backup copy made at the time the audit trail is created. (See the RECOVER TABLE statement.) See the section "Audit Trails" in Chapter 1 for more information.
2. If an audit trail file with the same pathname already exists, the CREATE AUDIT statement does nothing. If an audit trail file for the same table exists with a different pathname, **INFORMIX-ESQL/C** returns an error.

3. Make a backup copy of your database files as soon as you run the CREATE AUDIT statement, but before you make any further changes to the database. (See RECOVER TABLE for an example.) If possible, put the audit trail file on a different physical device from the one that holds your data, so that a failure of one does not damage the data on the other.
4. Audit trails slow INFORMIX-ESQL/C slightly because each alteration of the database is recorded in the audit trail file, as well as in the database files.
5. You must own *table-name* or have DBA status to use the CREATE AUDIT statement.
6. You must set execute permission for all directories below **root** in *pathname* for each class of user (owner, owner's group, and public) that accesses your database.
7. You cannot create a cluster index on a table with an audit trail.

### Example

---

```
$ create audit for orders in "/dev/safe/audord";
```

---

### Related Statements

DROP AUDIT, RECOVER TABLE

# CREATE DATABASE (O)

## Overview

Use the CREATE DATABASE statement to create a database. INFORMIX-ESQL/C creates the system catalogs that contain the data dictionary describing the structure of the database. The database you create automatically becomes the current database.

## Syntax

---

```
CREATE DATABASE database-name  
[WITH LOG IN "pathname" [MODE ANSI]]
```

---

## Explanation

CREATE  
DATABASE      are required keywords.

*database-name*      is the name you assign to the database. The *database-name* can be a host string variable containing the name of the database you want to create.

WITH LOG IN      are optional keywords.

*pathname*      is the full pathname of the transaction log file. The *pathname* must be enclosed in quotation marks.

MODE ANSI      are optional keywords.

## Notes

1. INFORMIX-ESQL/C creates a subdirectory in the current directory with the name *database-name.dbs*. All of the system catalogs, tables, data, and index files will be placed in this subdirectory, except for tables that you explicitly instruct INFORMIX-ESQL/C to create elsewhere.



2. A database name can be up to 10 characters in length and can contain only letters, digits, and underscores (\_). The first character must be a letter. If you store more than one database in a single directory, the database names must be unique.
3. For a user to have access to a database, the user must have execute (search) permission for each directory in the full pathname of *database-name.dbs*, as well as appropriate database privileges (see the GRANT statement).
4. See Appendix B for a detailed description of the system catalogs.
5. The WITH LOG IN clause creates a transaction log file. Without this file, you cannot use the BEGIN WORK, COMMIT WORK, or ROLLBACK WORK statements. You can use the START DATABASE statement to assign a log file to an existing database. See the section “Transactions” in Chapter 1 for further details.

You must include the WITH LOG IN keywords to use the MODE ANSI keywords in the CREATE DATABASE statement.

6. A database created as MODE ANSI supports implicit transactions. All statements automatically appear within a transaction. (Do not use the BEGIN WORK statement in a program that accesses a MODE ANSI database.) You explicitly terminate a transaction when you issue a COMMIT WORK or ROLLBACK WORK statement.
7. You cannot remove MODE ANSI from a database. Once created, a database remains MODE ANSI.
8. You can determine the type of database that a user selects by checking the warning flags following a DATABASE statement in the *sqlca.sqlwarn* structure. See the section “Error Handling and the *sqlca* Structure” in Chapter 2 for more information.
9. The CREATE DATABASE statement cannot appear in a multi-statement PREPARE.



## Examples

The following example creates the **stores** database:

---

```
$ create database stores  
    with log in "/s/log/stores.log";
```

---

The following example creates the **stores2** database in **MODE ANSI**:

---

```
$ create database stores2  
    with log in "/s/log/stores2.log" mode ansi;
```

---

## Related Statements

**DROP DATABASE, GRANT, START DATABASE**

# CREATE INDEX

## Overview

Use the CREATE INDEX statement to create an index for one or more columns in a table and optionally to cluster the physical table in the order of the index.

## Syntax

---

```
CREATE [UNIQUE] [CLUSTER] INDEX index-name  
ON table-name (column-name [ASC | DESC], ...)
```

---

## Explanation

CREATE INDEX	are required keywords.
UNIQUE	is a keyword you use to prevent duplicate entries in the column or the composite column to which the index applies.
CLUSTER	is an optional keyword that causes the physical table to be ordered according to the order of the index.
<i>index-name</i>	is the SQL identifier you want to assign to the index. You must assign a different identifier to each index in the database.
ON	is a required keyword.
<i>table-name</i>	is the name of the table containing the column or columns you want to index.

*column-name*

is the name of a column you want to index. To create an index that applies to several columns, enter a list of column names separated by commas. All the columns you specify must belong to the same table.

ASC

is a keyword that specifies an index that INFORMIX-ESQL/C maintains in ascending order. ASC is the default.

DESC

is a keyword that specifies an index that INFORMIX-ESQL/C maintains in descending order.

### Notes

1. When INFORMIX-ESQL/C executes the CREATE INDEX statement, it locks *table-name* IN EXCLUSIVE MODE. If another process is using *table-name*, INFORMIX-ESQL/C cannot execute CREATE INDEX and returns an error.
2. You can include up to eight columns in a composite index.
3. The total length of all columns indexed in a single CREATE INDEX statement cannot exceed 120 bytes.
4. See the section "Indexing Strategy" in Chapter 1 for a discussion of indexing strategy.
5. The CREATE CLUSTER INDEX statement fails if a CLUSTER index already exists.
6. You cannot CREATE CLUSTER INDEX on a table with an audit trail.
7. When there is more than one column listed, the concatenation of the set of columns is treated as a single composite column for the purpose of indexing.
8. The CREATE INDEX statement cannot be rolled back.

9. Descending indexes are not used to optimize queries. They are used to improve the performance of ORDER BY clauses in SELECT statements.
10. You can specify that a column or composite column allows only unique values when you create a table. You use the UNIQUE keyword in the CREATE TABLE statement for this purpose.
11. You cannot create an ascending index on a column defined as UNIQUE in a CREATE TABLE statement.
12. A column list defined as UNIQUE in a CREATE TABLE statement receives a unique ascending composite index. You cannot use the CREATE INDEX statement to create an identical unique composite index.
13. You can include in a composite index a column defined as UNIQUE. Similarly, you can include in a composite index a composite column list defined as UNIQUE. However, the column list in the CREATE INDEX statement cannot be identical to the column list defined as UNIQUE in the CREATE TABLE statement.
14. See the CREATE TABLE statement for more information about using the UNIQUE keyword to define UNIQUE CONSTRAINTS.
15. DISTINCT is a synonym for UNIQUE.

## Examples

---

```
$ create unique index i_ordnum
    on orders (order_num);

$ create cluster index i_ordnum2
    on orders (order_num desc);
```

---

## Related Statements

ALTER INDEX, DROP INDEX



# CREATE SYNONYM

## Overview

Use the CREATE SYNONYM statement to provide an alternative name for a table or view.

## Syntax

---

```
CREATE SYNONYM synonym FOR table-name
```

---

## Explanation

**CREATE  
SYNONYM** are required keywords.

*synonym* is an SQL identifier.

**FOR** is a required keyword.

*table-name* is the name of a table or view.

## Notes

1. In a database created as MODE ANSI, the name of a synonym is qualified by the owner of the synonym (*owner.synonym*). You must specify *owner* when you refer to a synonym owned by another user.

The use of the prefix *owner.* is optional in a non-MODE ANSI database. INFORMIX-ESQL/C does check the accuracy of *owner* if you include it in a statement, however. See the section "Owner Naming" in Chapter 1 for more information.

2. A user has no privileges under a synonym that were not granted for the table to which it applies.
3. When a synonym is created in an INFORMIX-ESQL/C program, the owner of the synonym is the person who runs the program.

4. Once created, a synonym persists until the owner of the synonym executes the DROP SYNONYM statement. This property distinguishes the synonym from the alias that you can use in the FROM clause of a SELECT statement. The alias persists only until the completion of the SELECT statement.
5. The CREATE SYNONYM statement cannot be rolled back.
6. For a database created as MODE ANSI, *owner.synonym* must be unique among all the synonyms, tables, and views in the database. In a non-MODE ANSI database, *synonym* must be unique.

### Example

---

```
$ create synonym cust for customer;
```

---

### Related Statements

DROP SYNONYM, SELECT

---

**Note:** Synonyms are very useful for referencing external objects with INFORMIX-OnLine. Refer to the *INFORMIX-OnLine Programmer's Manual* for more information.

---

# CREATE TABLE (O)

## Overview

Use the CREATE TABLE statement to create a new table in the current database.

## Syntax

---

```
CREATE [TEMP] TABLE table-name
    (column-name datatype
        [NOT NULL] [UNIQUE [CONSTRAINT constr-name]] [...]  
        [UNIQUE (unique-column-list) [CONSTRAINT constr-name]] [...])  
    [WITH NO LOG]  
    [IN pathname]
```

---

## Explanation

CREATE TABLE	are required keywords.
TEMP	is an optional keyword.
<i>table-name</i>	is the identifier you want to assign to the table. The first 10 characters must be unique within a database.
<i>column-name</i>	is the identifier you want to assign to each column.
<i>datatype</i>	specifies the data type for each column. (See the following list for valid data types.)
NOT NULL	are optional keywords.
UNIQUE	is an optional keyword specifying that the column or composite <i>unique-column-list</i> cannot contain duplicate values.
<i>unique-column-list</i>	lists the name of each column that you want to include in the composite UNIQUE CONSTRAINT.
CONSTRAINT	is a keyword you use to indicate that <i>constr-name</i> is assigned in the statement.

*constr-name*

is the name of the UNIQUE CONSTRAINT. *constr-name* must be a valid identifier that does not conflict with an existing constraint name. It can be optionally prefixed with the login name of the owner of the table or, if you have DBA privileges, the login name of another user.

WITH NO LOG

are optional keywords that prevent logging of TEMP tables when logging is in effect. In a database that uses logging, the default is to log TEMP tables also.

IN

is an optional keyword.

*pathname*

specifies the full pathname in which you want to store the database table, with no extension to the filename. The *pathname* cannot be longer than 64 characters, and must be contained within quotes (").

A pathname is of the form

*[/directory-name/...]filename*

A list of valid SQL data types follows:

CHAR(*n*)

is a character string of length *n* (where  $1 \leq n \leq 32,511$ ).

CHARACTER

is a synonym for CHAR.

SMALLINT

is a whole number from -32,767 to +32,767.

INTEGER

is a whole number from -2,147,483,647 to +2,147,483,647.

INT

is a synonym for INTEGER.

DECIMAL[(*m*[,*n*])]

is a decimal floating-point number with a total of *m* ( $\leq 32$ ) significant digits (precision) and *n* ( $\leq m$ ) digits to the right of the decimal point (scale). See the section "Database Data Types" in Chapter 1 for more information.

DEC

is a synonym for DECIMAL.



NUMERIC	is a synonym for DECIMAL.
SMALLFLOAT	is a binary floating-point number corresponding to the “float” data type in the C language.
REAL	is a synonym for SMALLFLOAT.
FLOAT[( <i>n</i> )]	is a binary floating-point number corresponding to the “double” data type in the C language. You can use <i>n</i> to specify the precision of a FLOAT data type, although the precision is ignored by INFORMIX-ESQL/C. <i>n</i> must be a whole number between 1 and 14.
DOUBLE PRECISION	is a synonym for FLOAT.
MONEY[( <i>m</i> [( <i>n</i> )])]	is a DECIMAL type number, displayed with leading \$. MONEY( <i>m</i> ) = DECIMAL( <i>m</i> ,2) and MONEY = DECIMAL(16,2). See the section “Database Data Types” in Chapter 1 for more information.
SERIAL[( <i>n</i> )]	is a sequential integer assigned automatically by SQL. You may assign an initial value <i>n</i> . The default starting integer is 1.
DATE	is a date entered as a character string in one of the formats described in the following notes.
DATETIME	is a moment in time that can include the year, month, day, hour, minute, second, and fraction of a second. See the following notes and the section “Database Data Types” in Chapter 1 for more information.
INTERVAL	is a span of time that can include years and months or days, hours, minutes, seconds, and fractions of a second. See the following notes and the section “Database Data Types” in Chapter 1 for more information.

## Notes

1. In a database created as MODE ANSI, the name of a table is qualified by the owner of the table (*owner.table-name*). You must specify *owner* when you refer to a table owned by another user.

The use of the prefix *owner.* is optional in a non-MODE ANSI database. INFORMIX-ESQL/C does check the accuracy of *owner* if you include it in a statement, however. See the section “Owner Naming” in Chapter 1 for more information.

2. If the database is MODE ANSI, the combination *owner.tablename* must be unique. Otherwise, table names must be unique within a database.
3. Column names must be unique within each table, but you can use duplicate names in different tables in the same database.
4. Temporary tables created with the TEMP option exist for the duration of the program.
5. Users with CONNECT privilege may create temporary tables.
6. The default value for a column is NULL unless you include the NOT NULL keywords after the data type of the column.
7. If you designate a column as NOT NULL, you must enter a value into this column when performing an INSERT or UPDATE to the table.
8. By default, all users who have been GRANTED CONNECT privilege to a non-MODE ANSI database have all access privileges to the new table except ALTER. To restrict access, use the REVOKE statement to take ALL access away from PUBLIC (everyone on the system), then use the GRANT statement to designate the access privileges you want to give to specific users.

In a database created as MODE ANSI, no default table-level privileges exist. You must explicitly GRANT these privileges.

9. You can specify no more than one SERIAL column in a table.
10. Enter DATE data types in the sequence of month, day, and year, with any non-numeric character, including a blank, as a separator. Represent the month as the number of the month (January = 1 or

01, February = 2 or 02, and so on). Represent the day as the day of the month (1 or 01, 2 or 02, and so on). The year is stored as a four-digit number (0001 to 9999). If you enter two digits yy for the year, SQL assumes the year is 19yy .

The following are all acceptable representations of June 1, 1989: 06/01/89, 6.1.89, and 6-1-1989.

11. The DATE type is actually stored as the integer number of days since December 31, 1899. You can sort DATE columns and make chronological comparisons between two DATE columns.
12. The following table shows the file space requirements (in bytes) for each data type:

SERIAL	4
SMALLINT	2
INTEGER	4
SMALLFLOAT	4
FLOAT	8
CHAR(n)	n
DECIMAL(m,n)	1 + m/2
MONEY(m,n)	1 + m/2
DATE	4
DATETIME	Depends on precision (see below)
INTERVAL	Depends on precision (see below)

A DATETIME column consists of a sequence of the following fields: year, month, day, hour, minute, second, and fraction(*n*). All fields of a DATETIME column are 2-digit integers, except for the year and fraction fields. The year field requires 4 digits. The fraction field requires *n* digits, rounded up to an even number. The number of bytes required for a DATETIME column is equal to half the total number of digits for all fields, plus 1.

AN INTERVAL column consists of a sequence of the following fields: year and month, or year, month, day, hour, minute, second, and fraction(*n*). All fields of an INTERVAL column are 2-digit integers, except for the first field and the fraction field. The number of digits in the first field is 2, unless otherwise specified as part of the qualifier. The fraction field requires *n* digits. The number of bytes required for an INTERVAL column is equal to half the total number of digits for all fields, rounded up to an even number, plus 1.

13. You cannot ROLLBACK the CREATE TABLE statement.



14. You can use the UNIQUE keyword to require that a single column or set of columns accept only unique data. A column or composite column list specified as UNIQUE is referred to as having a UNIQUE CONSTRAINT.
15. Each column in *unique-column-list* must be a column in the table and must not appear in the list more than once.
16. Each column name in *unique-column-list* must identify a column that has been declared as NOT NULL.
17. You cannot insert duplicate values into a UNIQUE column.
18. You cannot create an ascending index on a UNIQUE column. You cannot create an ascending composite index on an identical composite column list declared as UNIQUE.
19. You can include a UNIQUE column in a composite index created with the CREATE INDEX statement.
20. Use the ALTER TABLE statement to add or drop a UNIQUE CONSTRAINT from a column or composite column list. You can query the sysconstraints system catalog for the names of constraints.
21. You can include up to 8 columns in a *unique-column-list*. The total length of all the *column-names* in a *unique-column-list* cannot exceed 120 bytes.
22. If you do not specify a *constr-name*, INFORMIX-ESQL/C generates one using the template *u<tabid>\_<index number>*. If this name conflicts with an existing identifier, INFORMIX-ESQL/C returns an error and you must supply *constr-name*.
23. INFORMIX-ESQL/C implements the UNIQUE CONSTRAINT by creating a unique index for every column declared as UNIQUE in the CREATE TABLE statement. A row is added to the sysindexes file for each index. Each index name is created with the format *[space]<tabid>\_<index number>*.
24. If the *pathname* in an IN clause specifies a filename that is different from the *table-name*, always use the *table-name* (rather than the filename) to refer to the table in subsequent SQL statements.



25. The *pathname* in an IN clause can specify any valid directory and is not restricted to the directory that contains the current database. This enables a database to include tables in different disk volumes, or on different network nodes. Use this feature if your database is becoming too large for your current disk volume.
26. If you use the WITH NO LOG keywords in a CREATE TABLE statement and the database does not use logging, the WITH NO LOG option is ignored. You can use the WITH NO LOG option with a MODE ANSI database.
27. Once you turn off the logging on a temporary table, you cannot turn it back on again; a temporary table is therefore always logged or never logged.

## Examples

The sequence of statements that creates the **stores** database follows:

---

```

$ create database stores;

$ create table customer
(
    customer_num    serial(101),
    fname           char(15),
    lname           char(15),
    company          char(20),
    address1         char(20),
    address2         char(20),
    city             char(15),
    state            char(2),
    zipcode          char(5),
    phone            char(18)
);

$ create table orders
(
    order_num        serial(1001),
    order_date        date,
    customer_num      integer,
    ship_instruct     char(40),
    backlog           char(1),
    po_num            char(10),
    ship_date         date,
    ship_weight       decimal(8,2),
    ship_charge       money(6),
    paid_date         date
);

$ create table items
(
    item_num          smallint,
    order_num          integer,
    stock_num          smallint,
    manu_code          char(3),
    quantity           smallint,
    total_price        money(8)
);

$ create table stock
(
    stock_num          smallint,
    manu_code          char(3),
    description        char(15),
    unit_price         money(6),
    unit               char(4),
    unit_descr         char(15)
);

$ create table manufact
(
    manu_code          char(3),
    manu_name          char(15)
);

$ create table state
(
    code              char(2),
    sname              char(15)
);

```

---

The following statement creates the **tab1** table. In **tab1**, column **c1** is **UNIQUE** and the constraint is named **uc1**. A **UNIQUE CONSTRAINT** is also applied to the composite columns **c3** and **c4**.

---

```
$ create table tab1
  (c1 integer not null unique constraint uc1,
   c2 integer,
   c3 integer not null,
   c4 char(10) not null,
   unique (c3,c4));
```

---

The following statement creates the **employee** table. The data for the table is stored in the file **/a/work/employ.dat**. The index information is stored in the file **/a/work/employ.idx**.

---

```
$ create table employee
  (employ_num      serial(101),
   fname          char(15),
   lname          char(15),
   address        char(20),
   city           char(15),
   state          char(2),
   zipcode        char(5),
   phone          char(18),
   hire_date      date)
in "/a/work/employ";
```

---

The following example shows the use of the **DATETIME** and **INTERVAL** data types:

---

```
$ create table tv_programs (
  prog_title char(32),
  air_date datetime year to day not null,
  air_time datetime hour to minute,
  duration interval hour to second
);
```

---

The following example shows how to prevent logging of **TEMP** tables in a database that uses logging:

---

```
$ create temp table tab2 (fname char(15), lname char(15)) with no log;
```

---

## Related Statements

**ALTER TABLE, CREATE DATABASE, CREATE INDEX, DROP DATABASE, DROP TABLE, GRANT, REVOKE**

# CREATE VIEW

## Overview

Use **CREATE VIEW** to create a new view based upon existing tables and views in the database.

## Syntax

---

```
CREATE VIEW view-name [(column-list)]  
    AS SELECT-statement [WITH CHECK OPTION]
```

---

## Explanation

<b>CREATE VIEW</b>	are required keywords.
<i>view-name</i>	is an SQL identifier.
<i>column-list</i>	is a list of one or more identifiers that name the columns of <i>view-name</i> .
<b>AS</b>	is a required keyword.
<i>SELECT-statement</i>	is a <b>SELECT</b> statement.
<b>WITH CHECK OPTION</b>	are optional keywords.

## Notes

1. In a database created as **MODE ANSI**, the name of a view is qualified by the owner of the view (*owner.view-name*). You must specify *owner* when you refer to a view owned by another user.



The use of the prefix *owner.* is optional in a non-MODE ANSI database. INFORMIX-ESQL/C does check the accuracy of *owner* if you include it in a statement, however. See the section “Owner Naming” in Chapter 1 for more information.

2. Except for the statements in the following list, you can use a view in any SQL statement where you can use a table.

ALTER INDEX	DROP INDEX
ALTER TABLE	LOCK TABLE
CREATE INDEX	RENAME TABLE

The view behaves like a table with the name *view-name* and consists of the set of rows and columns returned by the *SELECT-statement* each time the SQL statement is executed using the view. The view reflects changes to the underlying tables with one exception.

If the view is defined with a *SELECT \** clause, it has only the columns in the underlying tables at the time the view is created. New columns added subsequently to the underlying tables using the *ALTER TABLE* statement will not appear in the view. See the section “Views” in Chapter 1 for more information.

3. When you do not specify *column-list* for *view-name*, the view inherits the column names of the underlying tables. If the *SELECT-statement* returns an expression, the corresponding column in the view is called a *virtual* column. You must provide a name for virtual columns. You must also provide a column name when the *select-list* has duplicate column names when the table prefixes are stripped. For example, when both **orders.order\_num** and **items.order\_num** appear in the *select-list*, you must provide two separate column names to label them in the *CREATE VIEW* statement.
4. Data types of the columns of the view are inherited from the tables from which they come. Data types of virtual columns are determined from the nature of the expression.
5. For a database created as *MODE ANSI*, *owner.view-name* must be unique among all the tables, views, and synonyms in the database. In a non-MODE ANSI database, *view-name* must be unique.

6. You can define a view in terms of other views, except that you must abide by the restrictions on queries listed in the section “Querying through Views” in Chapter 1.
7. The *SELECT-statement* cannot have an **ORDER BY** clause or a **UNION** operator, and it cannot include host variables.
8. You must have **SELECT** privilege on all columns from which the view is derived.
9. The **WITH CHECK OPTION** clause instructs **INFORMIX-ESQL/C** to ensure that all modifications to the underlying tables made through the view satisfy the definition of the view.
10. The **CREATE VIEW** statement cannot be rolled back.

### Example

---

```
$ create view palo_alto as
  select * from customer
    where city = "Palo Alto";
```

---

### Related Statements

**CREATE TABLE, DROP VIEW**

# DATABASE

## Overview

Use the DATABASE statement to select an accessible database as the current database.

## Syntax

---

**DATABASE** *database-name* [EXCLUSIVE]

---

## Explanation

**DATABASE** is a required keyword.

*database-name* is the name of the database you want to select. It can also be a host variable that evaluates to the name of a database.

**EXCLUSIVE** is an optional keyword.

## Notes

1. If you want to specify a database that does not reside in your current directory nor in a directory specified by the DBPATH environment variable (see Appendix C), you must follow the DATABASE keyword with a host variable that evaluates to the full pathname of the database (excluding the .dbs extension).
2. The EXCLUSIVE option opens the database IN EXCLUSIVE MODE and allows only the DBA access to the database. To allow others access to the database, you must execute the CLOSE DATABASE statement and then reopen the database.
3. The DATABASE statement cannot be rolled back.
4. This statement closes the current database, if one exists.

5. You can determine the type of database selected by checking the warning flags following a DATABASE statement in the **sqlca.sqlwarn** structure. See the section “Error Handling and the **sqlca** Structure” in Chapter 2 for more information.
6. The DATABASE statement cannot appear in a multi-statement PREPARE.

### Example

---

```
$ database stores;
```

---



# DECLARE

## Overview

Use the DECLARE statement to assign a cursor name to a SELECT or INSERT statement.

## Syntax

---

```
DECLARE cursor-name [SCROLL] CURSOR [WITH HOLD] FOR
    {SELECT-statement [FOR UPDATE [OF column-list]] |
     INSERT-statement | statement-id}
```

---

## Explanation

DECLARE	is a required keyword.
<i>cursor-name</i>	is an SQL identifier.
SCROLL	is an optional keyword and can be used only with a SELECT cursor.
CURSOR	is a required keyword.
WITH HOLD	are optional keywords you use to indicate that the cursor is not closed on termination of a transaction.
FOR	is a required keyword.
<i>SELECT-statement</i>	is a SELECT statement.
FOR UPDATE	are optional keywords.
OF	is an optional keyword.
<i>column-list</i>	is a list of column names from the tables listed in the FROM clause of the SELECT statement.

*INSERT-statement* is an INSERT statement.

*statement-id* is the identifier of an INSERT or SELECT statement that has been previously PREPARED.

## Notes

1. You must DECLARE a SELECT cursor before it can be used in an OPEN, FETCH, DELETE, UPDATE, or CLOSE statement. You must DECLARE an INSERT cursor before it can be used in an OPEN, PUT, FLUSH, or CLOSE statement.
2. SCROLL cursors, INSERT cursors, and cursors WITH HOLD are Informix extensions to ANSI standard syntax. You receive a warning if you include the SCROLL, INSERT, or WITH HOLD keywords in a program and you have defined the DBANSIWARN environment variable or you compile with the -ansi flag.
3. Unlike other cursors, a cursor WITH HOLD is not closed when you execute a COMMIT WORK or ROLLBACK WORK statement. You must explicitly CLOSE a cursor WITH HOLD. (A CLOSE DATABASE statement closes all cursors, including cursors WITH HOLD.)
4. The following rules apply when using cursor manipulation statements in a database with transactions:
  - You must OPEN and CLOSE a regular cursor that is FOR UPDATE and an INSERT cursor within a transaction.
  - All FLUSH and PUT statements must appear within a transaction.
  - Each UPDATE, INSERT, or DELETE action must take place within a transaction.
  - You can open and close a cursor WITH HOLD that is FOR UPDATE outside a transaction; all FETCHes using it, however, must take place within a transaction.

These requirements are automatically satisfied if the current database is a MODE ANSI database.

5. You should exercise caution when using a cursor WITH HOLD to update a table, or when opening a SCROLL cursor WITH HOLD. See the section “The Cursor WITH HOLD and Locks” in Chapter 1 for more information.
6. You must include the SCROLL keyword in the DECLARE statement for a SELECT cursor if you issue a FETCH PREVIOUS, FETCH PRIOR, FETCH LAST, FETCH FIRST, FETCH CURRENT, FETCH RELATIVE, or FETCH ABSOLUTE statement.
7. If you use a *statement-id* to identify a SELECT or INSERT statement, you must have previously PREPARED the statement.
8. The *cursor-name* has meaning from the point at which it is DECLARED to the end of the source file. This means that the DECLARE statement for a cursor must physically appear before any statement that makes reference to the cursor. It is not a global variable that can be used in a separate source file.
9. You may not use the FOR UPDATE clause in the DECLARE statement for a SELECT cursor that SCROLLS.
10. If you use the FOR UPDATE clause, the SELECT statement is limited to a single table.
11. You must use the FOR UPDATE clause in the DECLARE statement for a non-SCROLLing SELECT cursor if you will later use either the UPDATE or the DELETE statement with the WHERE CURRENT OF *cursor-name* option.
12. If you do not specify any columns in the FOR UPDATE clause of a DECLARE statement, you can update any column in a subsequent UPDATE WHERE CURRENT OF statement. If you do specify one or more columns in the FOR UPDATE clause, you can



update only those columns in a subsequent UPDATE WHERE CURRENT OF statement. When you specify the column names in the FOR UPDATE clause, INFORMIX-ESQL/C can usually perform the updates more quickly.

13. The columns in an OF *column-list* clause need not be in the *select-list* of the SELECT clause.
14. When you DECLARE a cursor FOR UPDATE, each FETCH executed on that cursor locks the FETCHed row IN EXCLUSIVE MODE. The lock is released when you execute the next FETCH statement or when you CLOSE the cursor. See the section “Locking” in Chapter 1 for a more detailed description of table-level and row-level locking. When you execute the actions within a transaction, the locked rows are released only when you issue a COMMIT WORK or ROLLBACK WORK statement.
15. You cannot DECLARE a cursor for an INSERT statement that contains an embedded SELECT statement.
16. If you DECLARE a cursor for an INSERT statement that includes a VALUES clause containing only constants, INFORMIX-ESQL/C does not create a buffer but merely keeps count of the number of inserts. Such INSERTs are never flushed as the result of a PUT statement. Flushing occurs when you issue a FLUSH or CLOSE cursor statement.



## Examples

---

```
$  declare scurs cursor for
    select * from customer;

$  declare ucurs cursor for
    select * from customer where customer_num > 110
    for update of fname, lname;

$  declare icurs cursor for
    insert into stock
    values ($stock_no, $man_code, $descr,
    $u_price, $unit, $u_desc);

$  declare scurs scroll cursor for
    select * from orders
    where customer_num = 104;

$  declare pcurs cursor with hold for
    select customer_num, lname, city
    from customer;
```

---

## Related Statements

CLOSE, DELETE, FETCH, FLUSH, OPEN, PREPARE, PUT,  
SELECT, UPDATE

# DELETE

## Overview

Use the DELETE statement to delete one or more rows from a table.

## Syntax

---

```
DELETE FROM table-name  
      [WHERE {condition | CURRENT OF cursor-name}]
```

---

## Explanation

DELETE FROM	are required keywords.
<i>table-name</i>	is the name of the table from which you want to delete rows.
WHERE	is a keyword.
<i>condition</i>	is a condition of a standard WHERE clause (see the SELECT statement for further information). INFORMIX-ESQL/C deletes all rows that satisfy the condition in the WHERE clause.
CURRENT OF	are keywords.
<i>cursor-name</i>	is the SQL identifier of a previously DECLARED and positioned cursor.

## Notes

1. When you create a database with transactions that is not MODE ANSI, each DELETE statement you execute is treated as a single transaction, even if you do not use the BEGIN WORK and COMMIT WORK or ROLLBACK WORK statements.

2. Each row affected by a DELETE statement within a transaction is locked for the duration of the transaction; therefore, a single DELETE statement that affects a large number of rows locks those rows until the entire operation is completed. If the number of rows affected is very large, you may exceed the limits placed by your operating system on the maximum number of simultaneous locks. If this occurs, you may want either to reduce the scope of the DELETE statement or to lock the entire table before executing the statement.

See the section "Locking" in Chapter 1 for a more detailed description of table-level and row-level locking in INFORMIX-ESQL/C.

3. To use the WHERE CURRENT OF option, you must have previously DECLARED the *cursor-name* with the FOR UPDATE option.
4. If you use the CURRENT OF option, DELETE removes the row of the active set at the current position of the cursor. The cursor is left pointing between the remaining rows and cannot be used for DELETE or UPDATE until it is repositioned with a FETCH statement.
5. The DESCRIBE statement sets **sqlca.sqlwarn.sqlwarn4** to W when a DELETE statement is PREPARED without a WHERE clause. SQL produces the same warning if you PREPARE a DELETE statement, but do not EXECUTE it.
6. If your database has transactions and you use the WHERE CURRENT OF clause, you must execute the DELETE statement within a transaction.

## Examples

---

```
$ delete from items
    where order_num = $onum;

$ delete from orders
    where current of query_cursor;
```

---

These statements remove all items belonging to the order number set in the host variable **onum** from the **items** table and remove the row from the **orders** table pointed to by the cursor **query\_cursor**.

## Related Statements

DECLARE, INSERT, UPDATE



# DESCRIBE

## Overview

DESCRIBE obtains information about a statement that has been PREPARED.

## Syntax

---

DESCRIBE *statement-id* INTO *sqlda-ptr*

---

## Explanation

DESCRIBE	is a required keyword.
<i>statement-id</i>	is the SQL identifier of a previously PREPARED statement.
INTO	is a required keyword.
<i>sqlda-ptr</i>	is a C variable declared as a pointer to an <b>sqlda</b> structure.

## Notes

1. DESCRIBE sets *sqlda-ptr* to point to a **sqlda** structure and describes there the data that is retrieved when *statement-id* is executed.
2. The DESCRIBE statement sets **sqlca.sqlwarn.sqlwarn4** to W when an UPDATE or DELETE statement is PREPARED without a WHERE clause.
3. DESCRIBE sets **sqlca.sqlcode** to 0 for a SELECT statement without an INTO TEMP clause and to a positive integer for any other SQL statement. You can test **sqlca.sqlcode** against the following constants that are defined as positive integers in the header file **sqltype.h** (see Appendix A).

---

Integer	SQL Statement
SQ_SELECT	SELECT
SQ_SELINTO	SELECT INTO TEMP
SQ_UPDATE	UPDATE
SQ_UPDCURR	UPDATE (with cursor)
SQ_UPDALL	UPDATE (no WHERE clause)
SQ_DELETE	DELETE
SQ_DELCURR	DELETE (with cursor)
SQ_DELALL	DELETE (no WHERE clause)
SQ_INSERT	INSERT
SQ_DATABASE	DATABASE
SQ_CREATEDB	CREATE DATABASE
SQ_CLSDB	CLOSE DATABASE
SQ_DROPDB	DROP DATABASE
SQ_CREATAB	CREATE TABLE
SQ_CTEMP	CREATE TEMP TABLE
SQ_DRPTAB	DROP TABLE
SQ_CREIDX	CREATE INDEX
SQ_DRPIDX	DROP INDEX
SQ_CREASYN	CREATE SYNONYM
SQ_DROPSYN	DROP SYNONYM
SQ_CREVIEW	CREATE VIEW
SQ_DROPVIEW	DROP VIEW
SQ_ALTER	ALTER TABLE
SQ_RENTAB	RENAME TABLE
SQ_RENCOL	RENAME COLUMN
SQ_GRANT	GRANT
SQ_REVOKE	REVOKE
SQ_LOCK	LOCK TABLE
SQ_UNLOCK	UNLOCK TABLE
SQ_STATS	UPDATE STATISTICS
SQ_BEGWORK	BEGIN WORK
SQ_COMMIT	COMMIT WORK
SQ_ROLLBACK	ROLLBACK WORK
SQ_STARTDB	START DATABASE
SQ_RFORWARD	ROLLFORWARD DATABASE
SQ_LDINSERT	FLUSH INSERT BUFFER
SQ_CREAUD	CREATE AUDIT
SQ_STRAUD	START AUDIT
SQ_STPAUD	STOP AUDIT
SQ_DRPAUD	DROP AUDIT
SQ_RECTAB	RECOVER TABLE
SQ_CHKTAB	CHECK TABLE
SQ_REPTAB	REPAIR TABLE
SQ_WAITFOR	SET LOCK MODE
SQ_ALTIDX	ALTER INDEX
SQ_EXPLAIN	SET EXPLAIN

---

## Example

---

```
$ describe state_id into q_desc;
```

---

## Related Statements

PREPARE, OPEN

# DROP AUDIT

## Overview

Use the DROP AUDIT statement to delete an audit trail file.

## Syntax

---

DROP AUDIT FOR *table-name*

---

## Explanation

DROP AUDIT FOR are required keywords.

*table-name* is the name of the table whose audit trail file you want to delete.

## Notes

1. Use the DROP AUDIT statement to remove the old audit trail file when you have made a backup of your database files. Use the CREATE AUDIT statement to start a new audit trail, and then back up the table. See the section “Audit Trails” in Chapter 1 for more information.
2. You must own *table-name* or have DBA status to use the DROP AUDIT statement.

## Example

---

```
$ drop audit for orders;
```

---

## Related Statements

CREATE AUDIT, RECOVER TABLE



# DROP DATABASE

## Overview

Use the DROP DATABASE statement to delete an entire database, including all system catalogs, indexes, and data.

## Syntax

---

```
DROP DATABASE {database-name | char-variable}
```

---

## Explanation

DROP DATABASE      are required keywords.

*database-name*      is the name of the database you want to delete.

*char-variable*      is a host variable of type CHAR containing the name of the database you want to delete.

## Notes

1. You must own all the tables in the database or have DBA status to run the DROP DATABASE statement successfully. Otherwise, INFORMIX-ESQL/C sets `sqlca.sqlcode` to a negative value and does nothing.
2. The DROP DATABASE statement does not remove the database directory if there are any files in the database directory other than those created for database tables and their indexes.
3. You are not allowed to drop the current database. You must execute the CLOSE DATABASE statement first.
4. The DROP DATABASE statement cannot be rolled back.

5. The DROP DATABASE statement cannot appear in a multi-statement PREPARE.

### **Example**

---

```
$ drop database stores;
```

---

### **Related Statements**

CREATE DATABASE, CLOSE DATABASE

# DROP INDEX

## Overview

Use the DROP INDEX statement to delete an index.

## Syntax

---

DROP INDEX *index-name*

---

## Explanation

DROP INDEX are required keywords.

*index-name* is the name of the index you want to delete.

## Notes

1. You must be the owner of the index or have DBA privilege to use the DROP INDEX statement.
2. The DROP INDEX statement cannot be rolled back.
3. You cannot use the DROP INDEX statement to drop a UNIQUE CONSTRAINT from a table.

## Example

---

```
$ drop index i_ordnum;
```

---

## Related Statements

CREATE INDEX, CREATE TABLE

# DROP SYNONYM

## Overview

Use the DROP SYNONYM statement to remove a previously defined synonym.

## Syntax

---

DROP SYNONYM *synonym*

---

## Explanation

DROP  
SYNONYM

are required keywords.

*synonym*

is an INFORMIX-ESQL/C identifier.

## Notes

1. You can drop a synonym only if you created it.
2. When you compile a program containing a synonym, the synonym is replaced by the real table identifier in the compiled program. If you subsequently drop the synonym, the compiled program will still run.
3. The DROP SYNONYM statement cannot be rolled back.



## Example

---

```
$ drop synonym cust;
```

---

## Related Statement

CREATE SYNONYM

# DROP TABLE

## Overview

Use the DROP TABLE statement to remove a table, along with its associated indexes and data.

## Syntax

---

DROP TABLE *table-name*

---

## Explanation

DROP TABLE are required keywords.

*table-name* is the name of the table you want to remove.

## Notes

1. When you remove a table, you also delete the data stored in it, the indexes on columns, any synonyms assigned to it, and any authorizations you have granted on the table. You also delete all views based on the table.
2. You cannot drop any of the system catalog tables.
3. You must be the owner of a table or have DBA privilege to use the DROP TABLE statement.
4. The DROP TABLE statement cannot be rolled back.

## Example

---

```
$ drop table customer;
```

---

## Related Statement

CREATE TABLE

# DROP VIEW

## Overview

Use the DROP VIEW statement to remove a view from the database.

## Syntax

---

DROP VIEW *view-name*

---

## Explanation

DROP VIEW      are required keywords.

*view-name*      is the identifier of a view.

## Notes

1. You can drop only those views that you have created. See the section “Views” in Chapter 1 for more information.
2. When you drop *view-name*, you also drop all views defined in terms of *view-name*.
3. The DROP VIEW statement cannot be rolled back.

## Example

---

```
$ drop view cust1;
```

---

## Related Statements

CREATE VIEW, DROP TABLE



# EXECUTE

## Overview

EXECUTE runs a previously PREPARED statement.

## Syntax

---

EXECUTE *statement-id* [USING {*input-list* | DESCRIPTOR *sqlda-ptr*}]

---

## Explanation

EXECUTE is a required keyword.

*statement-id* is an SQL identifier named in a previously PREPARED statement.

USING is a keyword.

*input-list* is a list of host variables to be substituted as values for the question marks (?) in the statement indicated by *statement-id*. Use this option when you know the number and data types of the PREPARED statement.

DESCRIPTOR is an optional keyword.

*sqlda-ptr* is a C variable pointer to an **sqlda** structure that describes the undefined values in the PREPARED statement.

## Notes

1. Once you have PREPARED an SQL statement, you can EXECUTE it as often as you desire.
2. To use the USING clause, you must know the number of the parameters of the PREPARED statement. The data type of each variable in *input-list* must be compatible with the value expected in the PREPARED statement for the corresponding parameter.

3. The host variables in *input-list* can be associated with indicator variables if the syntax of the statement corresponding to *statement-id* permits them.
4. You can EXECUTE a PREPARED SELECT only INTO TEMP. Use DECLARE with the OPEN, FETCH, and CLOSE statements to execute other PREPARED SELECT statements.

### Example

---

```
sprintf(s1,"%s",  
      "update orders set po_num = ?, order_date = ?");  
$   prepare statement_1 from s1;  
$   execute statement_1 using $po_num, $order_date;
```

---

### Related Statements

DECLARE, EXECUTE IMMEDIATE, PREPARE

# EXECUTE IMMEDIATE

## Overview

The EXECUTE IMMEDIATE statement combines three SQL statements: it PREPAREs, EXECUTEs, and FREEs an SQL statement with a single command. Any associated overhead is released immediately after the statement has been executed.

## Syntax

---

EXECUTE IMMEDIATE *string-spec*

---

## Explanation

EXECUTE            is a required keyword.

IMMEDIATE        is a required keyword.

*string-spec*        is a quoted string or a host variable containing an SQL statement.

## Notes

1. Do not use the EXECUTE IMMEDIATE statement for the following SQL statements:

CLOSE	FETCH
DECLARE	OPEN
DESCRIBE	PREPARE
EXECUTE	SELECT
EXECUTE IMMEDIATE	

2. The following restrictions apply to the statement contained in the *string-spec*:

- It cannot contain multiple SQL statements.
- It should not include an SQL statement prefix or terminator, such as a dollar sign or semicolon.
- It should not contain a host language comment.
- It should not be associated with an input or output host variable list or with a descriptor.

### **Example**

---

```
$ execute immediate "create database testdb";
```

---

### **Related Statements**

**EXECUTE, FREE, PREPARE**



# FETCH

## Overview

Use the FETCH statement to move the cursor to a new row in the active set and to retrieve the values from that row.

## Syntax

---

```
FETCH [NEXT | {PREVIOUS | PRIOR} | FIRST | LAST | CURRENT |  
      RELATIVE n | ABSOLUTE m ] cursor-name  
[INTO variable-list | USING DESCRIPTOR sqlda-ptr]
```

---

## Explanation

FETCH	is a required keyword.
NEXT	is an optional keyword indicating the next row in the active list. NEXT is the default.
PREVIOUS	is an optional keyword indicating the prior row in the active list.
PRIOR	is an optional keyword that is synonymous with PREVIOUS.
FIRST	is an optional keyword indicating the first row of the active list.
LAST	is an optional keyword indicating the last row of the active list.
CURRENT	is an optional keyword indicating the current row of the active list.
RELATIVE <i>n</i>	is an optional keyword indicating the <i>n</i> th row relative to the current cursor position in the active list. A negative number specifies before

	the current cursor position. <i>n</i> can be either a numeric literal or a host variable.
<b>ABSOLUTE <i>m</i></b>	is an optional keyword indicating the <i>m</i> th row in the active list. <i>m</i> can be either a numeric literal or a host variable.
<i>cursor-name</i>	is an SQL identifier of a previously DECLARED cursor.
<b>INTO</b>	is an optional keyword.
<i>variable-list</i>	is a list of host variables that contains the column values of the row pointed to by <i>cursor-name</i> .
<b>USING DESCRIPTOR</b>	are optional keywords.
<i>sqlda-ptr</i>	is a pointer to a <b>sqlda</b> structure that controls the storage of values corresponding to the columns or expressions returned by the SELECT statement associated with <i>cursor-name</i> .

## Notes

1. **FETCH NEXT** is the default condition.
2. You must first **DECLARE** a **SCROLL** cursor before issuing a **FETCH PRIOR**, **FETCH FIRST**, **FETCH LAST**, **FETCH CURRENT**, **FETCH RELATIVE *n***, or **FETCH ABSOLUTE *m*** statement.
3. If the **SELECT** statement associated with the cursor has an **INTO** clause, there must be no **INTO** clause in any **FETCH** statement referring to that cursor. If the **SELECT** statement has no **INTO** clause, the **FETCH** statement must have one.

4. When the cursor points to the last row in the active set, a subsequent **FETCH NEXT** causes **INFORMIX-ESQL/C** to return a “not found” code (**sqlca.sqlcode** = **SQLNOTFOUND**).
5. If you issue a **FETCH PRIOR** (**FETCH PREVIOUS**) statement when the cursor points to the first row in the active set, **INFORMIX-ESQL/C** returns a “not found” code (**sqlca.sqlcode** = **SQLNOTFOUND**).
6. When you execute a **FETCH ABSOLUTE *m*** statement where no *m*th row exists in the active set, **INFORMIX-ESQL/C** returns a “not found” code (**sqlca.sqlcode** = **SQLNOTFOUND**).
7. When you execute a **FETCH RELATIVE *n*** statement where no *n*th row exists in the active set, **INFORMIX-ESQL/C** returns a “not found” code (**sqlca.sqlcode** = **SQLNOTFOUND**).
8. **FETCH** does not lock a row unless the **DECLARE** statement contains a **SELECT** with a **FOR UPDATE** clause. It is possible to retrieve a row that is being **UPDATED** or **DELETED** by a concurrent process.
9. If the cursor was declared **FOR UPDATE** and the current database uses transactions, you must use this statement within a transaction (that is, following a **BEGIN WORK** statement). In a **MODE ANSI** database, all operations take place inside a transaction, so a **FETCH** may be done at any time.

## Examples

---

```
$  fetch query_curs into $cnum, $lname;  
$  fetch first query_curs into $cnum, $lname;
```

---

## Related Statements

**OPEN, CLOSE, DELETE, UPDATE**



# FLUSH

## Overview

Use the FLUSH statement to force INFORMIX-ESQL/C to insert the buffered rows into the database without closing the cursor.

## Syntax

---

**FLUSH** *cursor-name*

---

## Explanation

**FLUSH** is a required keyword.

*cursor-name* is the name of a cursor that has been DECLARED for an INSERT statement.

## Notes

1. The global variables **sqlca.sqlcode** and **sqlca.sqlerrd[2]** indicate the result of each FLUSH statement. If INFORMIX-ESQL/C successfully inserts the buffered rows into the database, it sets **sqlca.sqlcode** to zero and **sqlca.sqlerrd[2]** to the number of rows inserted. If INFORMIX-ESQL/C is unsuccessful in its attempt to insert the rows into the database, it sets **sqlca.sqlcode** to a negative number (specifically, the number of the error message) and sets **sqlca.sqlerrd[2]** to the number of rows successfully inserted into the database.
2. You can use the FLUSH statement to force the insertion. You cannot delay insertion by not using the FLUSH statement. INFORMIX-ESQL/C automatically flushes the buffer when it is full.
3. Insert cursors that contain only constants in the values clause are not buffered. Additional PUTs cannot fill the buffer and trigger an automatic FLUSH. INFORMIX-ESQL/C keeps a count of the number



of rows to be inserted into the database, and the database is updated only when you issue a **FLUSH** or **CLOSE** statement.

---

**Caution!** Exiting a program without closing the cursor leaves the buffer unflushed. Rows inserted into the buffer and remaining since the last flush are lost. Do not expect the end of program to close the cursor and flush the buffer.

---

### **Example**

---

```
$ flush icurs;
```

---

### **Related Statements**

**CLOSE, DECLARE, OPEN, PUT**

# FREE

## Overview

The **FREE** statement releases any resources that the database engine has allocated to a **PREPARED** statement or to an **OPENED** and **CLOSED** cursor.

## Syntax

---

**FREE** {*statement-id* | *cursor-name*}

---

## Explanation

<b>FREE</b>	is a required keyword.
<i>statement-id</i>	is the name of a statement that has been <b>PREPARED</b> .
<i>cursor-name</i>	is the name of a cursor whose <b>DECLARE</b> statement includes the keywords <b>SELECT</b> or <b>INSERT</b> .

## Notes

1. After you **FREE** *statement-id*, a cursor or **EXECUTE** statement cannot use it until you **PREPARE** it again.
2. If you **DECLARE** *cursor-name* **FOR SELECT** or **FOR INSERT**, a statement is automatically prepared when you **OPEN** the cursor and has no programmer-assigned *statement-id*. To release engine resources from that statement, use **FREE** *cursor-name*. Afterward, the cursor cannot be used unless you **OPEN** it again.
3. Do not **FREE** a *cursor-name* that was **DECLARED FOR** *statement-id*. Use instead **FREE** *statement-id*.

## Examples

---

```
$ free query_2;
```

```
$ free scurs;
```

---

## Related Statements

DECLARE, EXECUTE IMMEDIATE, PREPARE

# GRANT

## Overview

Use the GRANT statement to specify user access privileges to a database or to the tables in a database.

## Syntax

---

GRANT *tab-privilege* ON *table-name* TO {PUBLIC | *user-list*}  
[WITH GRANT OPTION] [AS *grantor*]

GRANT *db-privilege* TO {PUBLIC | *user-list*}

---

## Explanation

GRANT is a required keyword.

*tab-privilege* is one or more of the following table-level access types (multiple privileges must be separated by commas):

ALTER	adds or deletes columns or modifies data types of columns.
DELETE	deletes rows.
INDEX	creates indexes.
INSERT	inserts rows.
SELECT [( <i>cols</i> )]	retrieves data from specified columns.



UPDATE [(cols)]      changes values in specified columns.

ALL [PRIVILEGES]      is all of the above.

SELECT and UPDATE take column names as arguments, allowing you to specify columns that the user can select or update. Separate column names with commas.

The keyword PRIVILEGES following ALL is optional.

ON      is a required keyword.

*table-name*      is the name of the table for which you are granting access privileges.

TO      is a required keyword.

PUBLIC      is the keyword you use to specify access privileges for all users.

*user-list*      is a list of login names for the users to whom you are granting access privileges. You can enter one login name or a series of login names, separated by commas.

WITH GRANT      are optional keywords.

OPTION

AS      is an optional keyword.

*grantor*      is the username of the user issuing the GRANT statement.

*db-privilege*      is one of the following database-level access types:

CONNECT      allows access to database tables without permission to create permanent tables and indexes.

<b>RESOURCE</b>	allows access to database tables with permission to create permanent tables and indexes.
<b>DBA</b>	allows full database administrator privileges.

## Notes

1. You can grant privileges only on tables you create or on tables for which you have been given privileges **WITH GRANT OPTION**.
2. If you do not enter any column names, the **SELECT** or **UPDATE** access that you grant applies to all columns.
3. When you **GRANT** table-level privileges to another user using the **WITH GRANT OPTION** phrase, you give that user the power to **GRANT** the same privileges to another user.
4. The **CONNECT** privilege allows the recipient to interact with the existing tables to the extent that table-level privileges allow. The **CONNECT** privilege alone forbids the recipient from creating tables (except temporary tables) and indexes. The **CONNECT** recipient can create views.
5. The **RESOURCE** privilege includes the **CONNECT** privilege and adds the permission to create tables and indexes.
6. The **DBA** privilege includes the **RESOURCE** privilege, as well as the ability to alter system catalogs, to drop, start, and roll forward the database, and to grant and revoke **CONNECT**, **RESOURCE**, and **DBA** privileges to and from other users.
7. When you create a database, you are the Database Administrator and have **DBA** privileges.
8. A **DBA** can use the **AS** keyword to grant table-level privileges on behalf of another user.

9. The GRANT statement cannot be rolled back.
10. The most restrictive privileges always take precedence. For example, when you grant RESOURCE privileges to a user but do not grant INDEX privileges at the table level, that user is not able to create indexes for that table. Similarly, when you grant a user CONNECT privileges and table-level INDEX privileges, that user is still prevented from using the CREATE INDEX statement.

## Examples

The following statements grant all table-level privileges (except ALTER) to all users who have CONNECT privileges to the database:

---

```
$ grant all on customer to public;  
$ revoke alter on customer from public;
```

---

When you create a table in a database that is not MODE ANSI, ALL table-level privileges (except ALTER) are automatically GRANTED to all users (PUBLIC). To restrict access privileges at the table level, you must REVOKE all privileges and then GRANT specific ones:

---

```
$ revoke all on customer from public;  
$ grant all on customer to joe, mary;  
$ grant select (fname, lname, company, city)  
  on customer to public;
```

---

In a database created as MODE ANSI, no default table-level privileges exist. You must explicitly GRANT these privileges.

## Related Statement

### REVOKE

# INSERT

## Overview

Use the **INSERT** statement to insert one or more new rows into an existing table.

## Syntax

---

```
INSERT INTO table-name [(column-list)]  
           {VALUES (value-list) | SELECT-statement}
```

---

## Explanation

<b>INSERT INTO</b>	are required keywords.
<i>table-name</i>	is the name of the table to which you want to add rows.
<i>column-list</i>	are the names of the columns into which you want to insert data. You can enter one column name or a series of column names, separated by commas.
<b>VALUES</b>	is a keyword.
<i>value-list</i>	are the values that you want to insert into the columns you specified. You can enter one or more host variables or constants, separated by commas.
<i>SELECT-statement</i>	is a valid <b>SELECT</b> statement.



## Notes

1. **INFORMIX-ESQL/C** inserts data into the columns in the specified table in the order in which you enter column names. It inserts the first value you enter into the first listed column, the second value into the second listed column, and so on.
2. Entering column names is optional. If you omit them, **INFORMIX-ESQL/C** assumes the values are listed in the order in which the columns are listed in the **syscolumns** systems catalog. Unless you have subsequently used the **ALTER TABLE** statement to change the order, the order is the same as when the table was created.
3. **INFORMIX-ESQL/C** inserts the rows of data that result from the **SELECT** statement into the table, just as though you had entered them with the **VALUES** keyword.
4. You cannot use *table-name* in the **FROM** clause of the **SELECT** statement. The data must be selected from other tables.
5. You cannot include an **INTO TEMP** clause or an **ORDER BY** clause in the **SELECT** statement.
6. Although the values you insert do not have to be of the same data type as the columns themselves, they must be compatible. You can insert only **CHAR** data into **CHAR** columns and only numeric or character representation of numeric data into numeric columns.
7. Enter a zero (0) for a **SERIAL** column in the **INSERT** statement if you want **INFORMIX-ESQL/C** to insert the next **SERIAL** value for the table. Enter a nonzero value for a **SERIAL** column that does not duplicate a value already in the table, if you want **SQL** to use that value. When you enter a nonzero value for a **SERIAL** column that duplicates a value already in the table and if a unique index is defined on the column, this action produces an error and **sqlca.sqlcode** is set to a negative value.
8. You can use host variables in the list of values.

9. All constant CHAR, DATE, DATETIME, and INTERVAL values must be enclosed in quotation marks.
10. As an alternative to using the NULL keyword in *value-list*, you may use a host variable with a negative indicator variable.
11. When you create a database with transactions that is not MODE ANSI, but do not use the BEGIN WORK and COMMIT WORK or ROLLBACK WORK statements, each INSERT statement you execute is treated as a single transaction.
12. Each row affected by an INSERT statement within a transaction is locked for the duration of the transaction; therefore, a single INSERT statement that affects a large number of rows locks those rows until the entire operation is completed. If the number of rows affected is very large, you may exceed the limits placed by your operating system on the maximum number of simultaneous locks. If you exceed the limits, you may want either to reduce the number of rows affected by the INSERT statement or lock the entire table before executing the statement.

See the section "Locking" in Chapter 1 for a more detailed description of table-level and row-level locking in INFORMIX-ESQL/C.

---

**Caution!** INFORMIX-ESQL/C makes every possible effort to perform data conversion, including converting the character string "123" into the integer 123. When the data cannot be converted, however, INSERT stops. Unless you have created the database with transactions, all changes made up to the point where the error is encountered remain, but subsequent rows from the SELECT statement are not inserted. Data conversion also fails if the target data type cannot hold the value offered. For example, you cannot insert the integer 123456 into a SMALLINT.

---

## Example

---

```
$  insert into customer
    values (0, $f_name, $l_name, $comp,
           $addr1, $addr2, "Palo Alto"
           "CA", $zip, $phone);
```

---

## Related Statements

DELETE, SELECT

# LOCK TABLE

## Overview

Use the **LOCK TABLE** statement to prohibit access to a table by other users.

## Syntax

---

**LOCK TABLE** *table-name* **IN** {**SHARE** | **EXCLUSIVE**} **MODE**

---

## Explanation

**LOCK TABLE**      are required keywords.

*table-name*          is the name of the table you want to lock.

**IN**                    is a required keyword.

**SHARE**                is the keyword you use to give other users read access to the table, but to prevent them from modifying any of the data it contains.

**EXCLUSIVE**          is the keyword you use to prevent other users from having *any* access to the table.

**MODE**                is a required keyword.

## Notes

1. Only one lock can apply to a table at any given time. That is, if a user locks a table (in either **SHARE** or **EXCLUSIVE MODE**), no other user can lock that table in either mode until the first user unlocks it.



2. You can use the LOCK TABLE statement to override row-level locking during the transaction. Normally, each row of a table affected by a statement within a transaction is locked for the duration of the transaction.
3. During transactions that affect a large number of rows, you may exceed the limit placed by your operating system on the maximum number of locks. If you lock the entire table as soon as you begin the transaction, however, INFORMIX-ESQL/C does not lock each row in the table. You may want to use this strategy when you execute a transaction that affects a large number of rows or every row in a table.
4. See the section "Locking" in Chapter 1 for further explanation of table-level and row-level locking in INFORMIX-ESQL/C.

### Example

---

```
$ lock table orders in exclusive mode;
```

---

### Related Statement

UNLOCK TABLE

# OPEN

## Overview

Use the OPEN statement to establish the search criteria for a SELECT cursor and initialize the system for subsequent FETCHes, or to set up an insert buffer for an INSERT cursor that references program host variables.

## Syntax

---

```
OPEN cursor-name [USING {variable-list |  
                     DESCRIPTOR sqlda-ptr}]
```

---

## Explanation

OPEN	is a required keyword.
<i>cursor-name</i>	is an SQL identifier of a previously DECLARED cursor.
USING	is an optional keyword.
<i>variable-list</i>	is a list of host variables, separated by commas, corresponding to the “?” parameters in the query associated with a SELECT cursor.
DESCRIPTOR	is an optional keyword.
<i>sqlda-ptr</i>	is a pointer to an <b>sqlda</b> structure that points to the values corresponding to the “?” parameters in the SELECT statement associated with a SELECT cursor.

## Notes

1. If *cursor-name* is associated with a SELECT statement, the OPEN statement examines the contents of the host variables or the values pointed to by *sqllda-ptr* and, using these values for the parameters in the SELECT statement, establishes the search criteria for determining the logical set of rows that satisfies the WHERE clause. This set of rows is called the *active set*. It leaves the cursor in an open state and pointing before the first row of the active set.
2. The active set is a dynamic collection of rows. The active set is not fixed at the time that the cursor is OPENed.
3. Once the active set for a SELECT cursor is determined, the host variables are not reexamined until you reopen the cursor.
4. You cannot use indicator variables in an OPEN statement.
5. If a SELECT cursor is already open, an OPEN statement closes the cursor and reopens it, creating new selection or insertion criteria dependent upon the current values of the host variables.
6. If *cursor-name* is associated with an INSERT statement (rather than a SELECT statement), the OPEN statement cannot include a USING clause.
7. If you reopen an INSERT cursor that is already open, INFORMIX-ESQL/C flushes the INSERT buffer. The global variable *sqlca.sqlerrd[2]* is set to the number of rows successfully inserted into the database.
8. A cursor declared FOR UPDATE is called an update cursor. In a database that uses transactions, you cannot OPEN an UPDATE cursor outside a transaction unless it also was declared WITH HOLD. You can OPEN a non-UPDATE cursor, or one declared WITH HOLD, at any time.

A transaction begins with a **BEGIN WORK** statement and ends with a **COMMIT WORK** or **ROLLBACK WORK** statement. In a **MODE ANSI** database, no **BEGIN WORK** is required; all actions take place inside transactions. Thus, while this restriction applies to all databases, it has no effect on a **MODE ANSI** database.

## Examples

---

```
$ declare s_curs cursor for
      select * from orders;
$ open s_curs;
```

---

---

```
sprintf(select_1, "%s %s %s %s %s",
  "select o.order_num, sum(total_price)",
  "from orders o, items i",
  "where o.order_date > ? and o.customer_num = ?",
  "and o.order_num = i.order_num",
  "group by o.order_num");

$ prepare statement_1 from select_1;

$ declare q_curs cursor for statement_1;

$ open q_curs using $o_date, $c_num;
...
```

---

## Related Statements

**CLOSE, DECLARE, FETCH, FLUSH, PREPARE, PUT**



# PREPARE

## Overview

PREPARE preprocesses one or more SQL statements for later execution.

## Syntax

---

PREPARE *statement-id* FROM *string-spec*

---

## Explanation

PREPARE is a required keyword.

*statement-id* is an SQL identifier for the PREPARED string.

FROM is a required keyword.

*string-spec* is either a string constant enclosed in quotation marks or a host variable that has been defined as a character array or a pointer to a string. *string-spec* must contain one or more SQL statements.

## Notes

1. The statement described in *string-spec* cannot contain host variables. Use a question mark (?) in places where a data value will be supplied as an input value in an EXECUTE or OPEN statement. You cannot use question marks as placeholders for SQL identifiers such as database names, table names, column names, and user names.
2. When you prepare a SELECT statement for use with the DECLARE statement, *string-spec* can include a SELECT statement followed by a FOR UPDATE clause.

3. To **PREPARE** a multi-statement SQL string, you include all SQL statements in a single string. Delimit the individual statements with semicolons.

The single **PREPARE** statement prepares each SQL statement in the string for execution; the single **EXECUTE** statement runs the previously **PREPARED** string.

See the section “Examples” for an example of this approach.

4. There is little functional difference between a multi-statement **PREPARE** and the preparation and execution of individual SQL statements. However, you can realize a performance improvement if you reduce the number of times the user program and the database engine must communicate.

Note that a multi-statement **PREPARE** is processed as a unit; actions are not treated sequentially as is done when executing an SQL command file. Therefore, you cannot include statements that are dependent upon actions that occur in a previous statement in the list.

5. Do not use the **PREPARE** statement on the following statements:

<b>CLOSE</b>	<b>FETCH</b>
<b>DECLARE</b>	<b>OPEN</b>
<b>DESCRIBE</b>	<b>PREPARE</b>
<b>EXECUTE</b>	<b>WHENEVER</b>

The **DATABASE**, **CREATE DATABASE**, **DROP DATABASE**, and **CLOSE DATABASE** statements cannot appear in a multi-statement **PREPARE**.

6. Do not include a **SELECT** statement in a multi-statement **PREPARE** or **PREPARE** a **SELECT** statement with an **INTO** clause. You can include a **SELECT** in a subquery in an **UPDATE** or **DELETE** operation, however.
7. **INFORMIX-ESQL/C** returns error status information on the first error it encounters in a multi-statement **PREPARE**. The programmer is responsible for identifying which SQL statement is in error. For this reason, you may want to prepare statements individually, and

then use a multi-statement PREPARE once the statements are debugged and increased program performance is desired.

8. The scope of *statement-id* is the module (source file) in which it is prepared. You can refer to it by name in two or more different functions contained in the same module. You cannot reference *statement-id* in a separate module.
9. Within a module, *statement-id* can apply to only one string constant. Do not prepare another string with the same *statement-id*.

## Examples

The first example PREPAREs a single SELECT statement and assigns the *statement-id* **query\_2**.

---

```
$  prepare query_2 from
    "select * from orders
      where customer_num = ? and
        order_date > ?";
```

---

The following example PREPAREs six SQL statements (expressed as nine C literal character strings) into a single string named **query** using **sprintf( )**. The individual statements are delimited with semicolons. A single **\$prepare** statement PREPAREs all six statements for execution, and a single **\$execute** statement is used to execute the PREPARED **qid** string.

---

```
printf(query, "%s %s %s %s %s %s %s %s %s",
    "begin work;",
    "update account set balance = balance + ? where",
    "acct_number = ? and balance + ? >= 0;",
    "update teller set balance = balance + ? where",
    "teller_number = ?;",
    "update branch set balance = balance + ? where",
    "branch_number = ?;",
    "insert into history values(timestamp, values);",
    "commit work;" );
$prepare qid from $query;
$execute qid using $delta, $acct_number, $delta, $delta,
    $teller_number, $delta, $branch_number;
```

---

The database engine will stop on the first error it finds and will return error status information relating to that first error. Thus, multi-statement lists should not be used when there is any doubt about the correctness of the syntax or logic.

### **Related Statements**

**DECLARE, EXECUTE, EXECUTE IMMEDIATE, OPEN**



# PUT

## Overview

Use the PUT statement to store a row in the INSERT buffer for later insertion into the database table.

## Syntax

---

```
PUT cursor-name  
  [{USING DESCRIPTOR sqlda-ptr | FROM host-variable-list}]
```

---

## Explanation

- |                      |  |
|----------------------|--|
| PUT                  | is a required keyword.   |
| <i>cursor-name</i>   | is the name of a cursor that has been DECLARED for an INSERT statement.  |
| USING<br>DESCRIPTOR  | are optional keywords.   |
| <i>sqlda-ptr</i>     | is a pointer to an <b>sqlda</b> structure that points to the values corresponding to the “?” parameters in the prepared INSERT statement associated with the cursor. |
| FROM                 | is an optional keyword.  |
| <i>variable-list</i> | is a list of host variables, separated by commas, corresponding to the “?” parameters in the PREPARED INSERT statement associated with the cursor.                   |

## Notes

1. You can execute the PUT statement only if *cursor-name* has been DECLARED for an INSERT statement and is in an open state. Such a cursor is referred to as an insert cursor.
2. The PUT statement puts a row in the buffer that was created when *cursor-name* was OPENED. When you flush the buffer (by executing a series of PUT statements, a CLOSE statement, or a FLUSH statement) INFORMIX-ESQL/C inserts the buffered rows into the database table as a block.
3. Insert cursors that contain only constants in the values clause are not buffered. Additional PUTs cannot fill the buffer and trigger an automatic FLUSH. INFORMIX-ESQL/C keeps a count of the number of rows to be inserted into the database, and the database is updated only when you issue a FLUSH or CLOSE statement.
4. You close a cursor by issuing a CLOSE statement. Exiting a program without closing an insert cursor leaves the buffer unflushed. Rows inserted into the buffer and remaining since the last flush are lost. You cannot rely on the end of program to close the cursor and flush the buffer.
5. The global variables `sqlca.sqlcode` and `sqlca.sqlerrd[2]` indicate the result of each PUT statement.
  - If INFORMIX-ESQL/C simply puts a row in the insert buffer, it sets both `sqlca.sqlcode` and `sqlca.sqlerrd[2]` to zero.
  - If, as the result of a PUT statement, INFORMIX-ESQL/C successfully inserts a block of rows into the database, it sets `sqlca.sqlcode` to zero and `sqlca.sqlerrd[2]` to the number of rows inserted.
  - If, as the result of a PUT statement, INFORMIX-ESQL/C is unsuccessful in its attempt to insert an entire block of rows into the database, it sets `sqlca.sqlcode` to a negative number (specifically, the number of the error message) and sets

`sqlca.sqlerrd[2]` to the number of rows successfully inserted into the database.

Buffered rows following the last successfully inserted row are discarded.

6. If your database has transactions (and is not MODE ANSI), you must execute a PUT statement within a BEGIN WORK - COMMIT WORK block or a BEGIN WORK - ROLLBACK WORK block. COMMIT WORK and ROLLBACK WORK automatically close all cursors, except cursors WITH HOLD.
7. If *cursor-name* has been DECLARED for a prepared INSERT statement that includes "?" parameters, you must use the PUT statement with a FROM or USING DESCRIPTOR clause. After the FROM keyword, you can list the host variable(s) containing the value(s) INFORMIX-ESQL/C substitutes for the "?" parameters in the prepared INSERT statement. After the USING DESCRIPTOR keywords, you can specify the name of a pointer to an `sqllda` structure that points to the value(s) for the "?" parameters.

## Examples

---

```
$ declare icurs cursor for
      insert into manufact values ($m_code, $m_name);
$ open icurs;
$ put icurs;
```

---

---

```
$ prepare ins_stmt from
      "insert into manufact values (?, ?)";
$ declare ins_curs cursor for ins_stmt;
$ open ins_curs;
$ put ins_curs from $m_code, $m_name;
```

---

## Related Statements

CLOSE, DECLARE, FLUSH, OPEN



# RECOVER TABLE

## Overview

In the event of a system failure, use the **RECOVER TABLE** statement to restore a database table from a backup copy of the table with an audit trail file.

## Syntax

---

**RECOVER TABLE** *table-name*

---

## Explanation

**RECOVER TABLE**    are required keywords.

*table-name*            is the name of the table you want to recover.

## Notes

1. Once you have recovered the table, use the **DROP AUDIT** statement to remove the contents of the audit trail file. Run the **CREATE AUDIT** statement to start a new audit trail file, then back up the table. See the section “Audit Trails” in Chapter 1 for more information.
2. **RECOVER TABLE** checks that the audit trail and *table-name* have consistent record numbers for rows where changes have taken place. If **RECOVER TABLE** finds inconsistencies, it stops restoring the table.
3. You must own *table-name* or have DBA status to use the **RECOVER TABLE** statement.



## Example

The following list of SQL statements shows a template for the recovery of a table. They assume that your audit trail began from the last backup.

---

```
{restore table from last backup}

$  recover table customer;

$  drop audit for customer;

$  create audit for customer in "/dev/safe/audord";

{make a backup of the recovered table}
```

---

## Related Statements

CREATE AUDIT, DROP AUDIT

# RENAME COLUMN

## Overview

Use the **RENAME COLUMN** statement to change the name of a column.

## Syntax

---

```
RENAME COLUMN table.oldcolumn TO newcolumn
```

---

## Explanation

**RENAME COLUMN** are required keywords.

*table* is the name of the table containing the column whose name you want to change. This is a required item in the statement.

*oldcolumn* is the name of the column you want to rename.

**TO** is a required keyword.

*newcolumn* is the new name you want to assign to the column. *newcolumn* must satisfy the requirements for an SQL identifier and cannot duplicate another column name in the table.

## Notes

1. You can **RENAME** a column of a table only when you own the table, have DBA privilege, or have been granted **ALTER** permission.
2. The **RENAME COLUMN** statement cannot be rolled back.

## Example

---

```
$  rename column customer.customer_num to c_num;
```

---

## Related Statements

**ALTER TABLE, CREATE TABLE, INSERT, DROP TABLE, RENAME TABLE**

# RENAME TABLE

## Overview

Use the RENAME TABLE statement to change the name of a table.

## Syntax

---

```
RENAME TABLE oldname TO newname
```

---

## Explanation

RENAME TABLE      are required keywords.

*oldname*              is the name of the table you want to rename.

TO                    is a required keyword.

*newname*            is the new name you want to assign to the table.

## Notes

1. You can RENAME a table only when you own the table, have DBA privilege, or have been granted ALTER permission on the table.
2. The RENAME TABLE statement cannot be rolled back.



## Example

This example moves the **quantity** column to the third place.

---

```
$ create table mytab
  (item_num      smallint,
   order_num     integer,
   quantity      smallint,
   stock_num     smallint,
   manu_code     char(4),
   total_price   money(8)
  );
$ insert into mytab
  select item_num, order_num,
         quantity, stock_num, manu_code,
         total_price from items;
$ drop table items;
$ rename table mytab to items;
```

---

## Related Statements

**ALTER TABLE, CREATE TABLE, INSERT, DROP TABLE, RENAME COLUMN**

# REVOKE

## Overview

Use the REVOKE statement to remove another user's access privileges for a database or table.

## Syntax

---

```
REVOKE {tab-privilege ON table-name | db-privilege}  
FROM {PUBLIC | user-list}
```

---

## Explanation

**REVOKE** is a required keyword.

*tab-privilege* is one or more of the following table-level access privileges, separated by commas:

ALTER	adds or deletes columns or modifies data types of columns.
DELETE	deletes rows.
INDEX	creates indexes.
INSERT	inserts rows.
SELECT	retrieves data.
UPDATE	changes column values.
ALL [PRIVILEGES] is all of the above.	

<b>ON</b>	is a required keyword.						
<i>table-name</i>	is the name of the table for which you are revoking access privileges.						
<i>db-privilege</i>	is one of the following database-level access types: <table> <tr> <td><b>CONNECT</b></td><td>allows access to database tables without permission to create permanent tables and indexes.</td></tr> <tr> <td><b>RESOURCE</b></td><td>allows access to database tables with permission to create permanent tables and indexes.</td></tr> <tr> <td><b>DBA</b></td><td>allows full database administrator privileges.</td></tr> </table>	<b>CONNECT</b>	allows access to database tables without permission to create permanent tables and indexes.	<b>RESOURCE</b>	allows access to database tables with permission to create permanent tables and indexes.	<b>DBA</b>	allows full database administrator privileges.
<b>CONNECT</b>	allows access to database tables without permission to create permanent tables and indexes.						
<b>RESOURCE</b>	allows access to database tables with permission to create permanent tables and indexes.						
<b>DBA</b>	allows full database administrator privileges.						
<b>FROM</b>	is a required keyword.						
<b>PUBLIC</b>	is the keyword you use to revoke access privilege from all users.						
<i>user-list</i>	is a list of login names for the users whose access privilege you are revoking. You can enter one login name or a series of login names, separated by commas.						

## Notes

1. The REVOKE statement cannot be rolled back.
2. You can revoke database-level access privileges only if you have DBA status.
3. You can revoke only table-level access privileges that you have GRANTED to another user.
4. You cannot revoke privileges from yourself.

5. Although you can grant UPDATE and SELECT privileges for specific columns, you cannot revoke these privileges column by column. If you revoke UPDATE or SELECT privileges from a user, INFORMIX-ESQL/C automatically revokes all UPDATE and SELECT privileges you have ever granted to that user for *table-name*. You can then GRANT privileges for specific columns.
6. Only a DBA recipient can revoke the DBA privilege from another recipient. If the database creator grants DBA privileges to another user, that person can revoke the DBA privilege from the database creator.
7. If you revoke the DBA or RESOURCE privilege from one or more users, they are left with the CONNECT privilege. To revoke all database privileges from users with DBA or RESOURCE status, you must revoke CONNECT as well as DBA or RESOURCE.

### Example

---

```
$  revoke delete, update  
    on customer from john, mary;
```

---

### Related Statement

GRANT



# ROLLBACK WORK

## Overview

Use the ROLLBACK WORK statement to undo all database modifications since the beginning of the transaction.

## Syntax

---

ROLLBACK WORK

---

## Explanation

ROLLBACK WORK are required keywords.

## Notes

1. The ROLLBACK WORK statement rolls back all database modifications made since the transaction was initiated.
2. See the section “Transactions” in Chapter 1 for more information.

## Example

---

```
$ rollback work;
```

---

## Related Statements

BEGIN WORK, COMMIT WORK

# ROLLFORWARD DATABASE

## Overview

Use the ROLLFORWARD DATABASE statement to cause INFORMIX-ESQL/C to apply the transactions in the transaction log file to a backup copy of your database, recovering all completed transactions.

## Syntax

---

ROLLFORWARD DATABASE *database-name*

---

## Explanation

ROLLFORWARD      are required keywords.  
DATABASE

*database-name*      is the name of a database.

## Notes

1. Immediately after the database is rolled forward, it is IN EXCLUSIVE MODE with no transactions. Once the database has been closed and reopened, it is accessible to users and transactions are resumed.
2. See the section "Transactions" in Chapter 1 for more information.

## Related Statements

BEGIN WORK, COMMIT WORK, START DATABASE, ROLLBACK WORK

# SELECT

## Overview

Use the **SELECT** statement to query the current database.

The **SELECT** statement can include up to eight clauses. Only the **SELECT** clause and the **FROM** clause are required.

## Syntax

---

```
SELECT clause [INTO clause] FROM clause  
              [WHERE clause]  
              [GROUP BY clause]  
              [HAVING clause]  
              [ORDER BY clause]  
              [INTO TEMP clause]
```

---

See the section “The **SELECT** Statement” later in this chapter for detailed descriptions of these clauses.

# SET EXPLAIN

## Overview

Use the SET EXPLAIN statement to record how the query processor is accessing the database when executing a query.

## Syntax

---

```
SET EXPLAIN {ON | OFF}
```

---

## Explanation

SET EXPLAIN	are required keywords.
ON	is an optional keyword that turns on the SET EXPLAIN command.
OFF	is an optional keyword that turns off the SET EXPLAIN command. OFF is the default.

## Notes

1. When you issue SET EXPLAIN ON, the access procedures of all subsequent queries are stored in the file **sqexplain.out** in your current directory. If **sqexplain.out** already exists, subsequent output is appended to it. SET EXPLAIN ON remains in effect until you issue SET EXPLAIN OFF, or the program ends.
2. SET EXPLAIN estimates the cost in CPU resources (a weighted sum of disk accesses and total rows processed), indicates the order of table access, and estimates the number of rows returned. For each table, SET EXPLAIN identifies the type of access and the column(s) that serves as a filter, including whether the filtering is through an index. The following table-access types are available:



SEQUENTIAL SCAN reads rows in sequence.

INDEX PATH scans one or more indexes.

AUTOINDEX PATH creates a temporary index.

3. The table-owner's name precedes table names in the output file.

## Examples

The following sample output shows an **sqexplain.out** file for a simple and a complex query from one table.

---

QUERY:

-----  
select fname, lname, company from customer;

Estimated Cost: 4

Estimated # of Rows Returned: 18

1) joe.customer: SEQUENTIAL SCAN

QUERY:

-----  
select fname, lname, company from customer  
where company matches "Sport\*" and customer\_num between 110 and 115  
order by lname;

Estimated Cost: 3

Estimated # of Rows Returned: 1

Temporary Files Required For: Order By

1) joe.customer: INDEX PATH

Filters: joe.customer.company MATCHES 'Sport\*'

(1) Index Keys: customer\_num

Lower Index Filter: joe.customer.customer\_num >= 110

Upper Index Filter: joe.customer.customer\_num <= 115

---

The following sample output is from an **sqexplain.out** file for a multiple-table query.

---

QUERY:

-----

```
select * from customer, orders, items where
customer.customer_num = orders.customer_num and
orders.order_num = items.order_num;
```

Estimated Cost: 110

Estimated # of Rows Returned: 41

1) joe.orders: SEQUENTIAL SCAN

2) joe.customer: INDEX PATH

(1) Index Keys: customer\_num

Lower Index Filter: joe.customer.customer\_num = joe.orders.customer\_num

3) joe.items: INDEX PATH

(1) Index Keys: order\_num

Lower Index Filter: joe.items.order\_num = joe.orders.order\_num

---

## Related Statement

### SELECT

---

**Note:** Additional statistics are available for query processing when you use **INFORMIX-OnLine** as the database engine. As a result, estimates for the cost and the number of rows returned may be more precise under **INFORMIX-OnLine**.

---

# SET LOCK MODE (O)

## Overview

Use the SET LOCK MODE statement to determine whether or not subsequent INFORMIX-ESQL/C calls wait for a locked row to become unlocked.

## Syntax

---

SET LOCK MODE TO [NOT] WAIT

---

## Explanation

SET LOCK MODE      are required keywords.

TO                    is a required keyword.

NOT                  is an optional keyword.

WAIT                is a required keyword.

## Notes

1. The TO NOT WAIT option causes INFORMIX-ESQL/C to return an error if a statement attempts to alter or delete a row (or to SELECT a row FOR UPDATE) that another process has locked. This is the default situation; that is, if you have not issued a SET LOCK MODE statement previously. The NOT option is relevant, therefore, only when you have previously executed SET LOCK MODE TO WAIT and want to return to the default state.
2. The TO WAIT option causes INFORMIX-ESQL/C to wait on an attempt to alter or delete a row (or to SELECT a row FOR UPDATE) that another process has locked until the locked row becomes unlocked.

3. Use the **SET LOCK MODE TO WAIT** statement with extreme caution. If the locking process fails and does not remove the lock, your statement could wait indefinitely.
4. This feature is available only on systems that have record-level locking and applies only to row-level locking. An attempt to access a row in a table that is locked **IN EXCLUSIVE MODE** returns an error.
5. You can use **SET LOCK MODE** only on systems that have kernel locking. To find out if your system has kernel locking, check the directory that holds the database files to see if it contains any files with the extension **.lok**. If that directory contains **.lok** files, your system does not have kernel locking and you cannot use the **SET LOCK MODE** command.



# START DATABASE

## Overview

Use the START DATABASE statement to start a new transaction log file, or to start a database that complies with ANSI standards.

## Syntax

---

```
START DATABASE database-name WITH LOG IN "pathname" [MODE ANSI]
```

---

## Explanation

START  
DATABASE      are required keywords.

*database-name*    is the name of a database.

WITH LOG IN    are required keywords.

*pathname*      is the full pathname of the transaction log file. The *pathname* must be enclosed in quotation marks.

MODE ANSI      are optional keywords.

## Notes

1. Use the START DATABASE statement to perform the following tasks:
  - Change the name of your transaction log file.
  - Start recording transactions in a database that was created without transactions.
  - Start a database that supports ANSI standards.

2. The **START DATABASE** statement opens the database IN **EXCLUSIVE MODE**. No users can access the database until you issue a **CLOSE DATABASE** statement.
3. A database started as **MODE ANSI** supports implicit transactions and the *owner.object* naming convention.

In a **MODE ANSI** database, you receive an error if you do not use the *owner.object* naming convention to refer to an object owned by another user. You should modify existing queries that reference a table, view, or synonym owned by another user to include the *owner* prefix. See the section “Owner Naming” in Chapter 1 for more information.

4. Do not use the **BEGIN WORK** statement in programs that access a **MODE ANSI** database. Since transactions are implicit in **MODE ANSI**, the **BEGIN WORK** statement is not needed.
5. Singleton transactions do not exist in **MODE ANSI**. You must issue a **COMMIT WORK** statement to commit a transaction, or a **ROLLBACK WORK** statement to roll the database back to the last **COMMIT WORK** or **ROLLBACK WORK** statement.
6. You cannot remove **MODE ANSI** from a database. Once started as such, a database remains **MODE ANSI**.
7. See the section “Transactions” in Chapter 1 for more information on **START DATABASE**. See the discussion of **CREATE DATABASE** earlier in this chapter for more information on **MODE ANSI** databases.
8. You can determine the type of database that a user selects by checking the warning flags following a **DATABASE** statement in the **sqlca.sqlwarn** structure. See the section “Error Handling and the **sqlca** Structure” in Chapter 2 for more information.

## Example

---

```
$ start database stores  
    with log in "/u/myname/stores.log" mode ansi;
```

---

## Related Statements

BEGIN WORK, COMMIT WORK, CREATE DATABASE, ROLLBACK  
WORK, ROLLFORWARD DATABASE

# UNLOCK TABLE

## Overview

Use the UNLOCK TABLE statement to unlock a table that you previously locked with the LOCK TABLE statement.

## Syntax

---

UNLOCK TABLE *table-name*

---

## Explanation

UNLOCK TABLE      are required keywords.

*table-name*              is the name of the table you want to unlock.

## Note

Within a transaction, an UNLOCK TABLE statement generates an error. If you have issued a LOCK TABLE statement in a MODE ANSI database, you must release the lock by issuing the COMMIT WORK or ROLLBACK WORK statement.

## Related Statement

LOCK TABLE



# UPDATE

## Overview

Use the UPDATE statement to change the values in one or more columns of one or more rows in a table.

## Syntax

---

```
UPDATE table-name SET {column-name = expr[, ... ] |  
  {(column-list) | *} = (expr-list)}  
  [WHERE {condition | CURRENT OF cursor-name}]
```

---

## Explanation

UPDATE	is a required keyword.
<i>table-name</i>	is the name of the table that contains the row(s) you want to update.
SET	is a required keyword.
<i>column-name</i>	is the name of a column you want to update.
<i>expr</i>	is any combination of column names, constants, host variables, arithmetic operators, or an SQL subquery that returns a single row of one value.
<i>column-list</i>	is a list of the names of the columns you want to update.
*	refers to all columns in <i>table-name</i> .
<i>expr-list</i>	is a list of expressions that represent values corresponding to the columns in <i>column-list</i> or the columns represented by the asterisk notation. The list can include an SQL subquery that returns a single row of multiple values.

**WHERE** is an optional keyword.

*condition* is a condition for a standard WHERE clause made up of a search condition that compares the values in one column to the values in another column, to a host variable, or to a constant. (For further information, refer to the explanation of WHERE clauses in the section "The SELECT Statement," later in this chapter.)

**CURRENT OF** are keywords.

*cursor-name* is the SQL identifier of a previously DECLARED cursor.

## Notes

1. *expr* can be a SELECT statement in parentheses that adheres to standard rules for subqueries. The SELECT statement can return no more than one value except when included in an *expr-list*.
2. You cannot use a SELECT statement that retrieves data from *table-name*.
3. The number of column names included in the *column-list* must equal the number of values produced in the *expr-list*.
4. Although the value returned by *expr* does not have to be of the same data type as *column-name*, it must be compatible with the *column-name*. You can put only CHAR data into CHAR columns and only numeric or character representations of numeric data into numeric columns.
5. If you use the CURRENT OF option in the WHERE clause, INFORMIX-ESQL/C updates the current row of the active set and leaves the cursor on the same row.
6. If you do not specify any columns in the FOR UPDATE clause of a DECLARE statement, you can update any column in a subsequent UPDATE WHERE CURRENT OF statement. If you do specify one or more columns in the FOR UPDATE clause, you can update only

those columns in a subsequent UPDATE WHERE CURRENT OF statement. When you specify the column names in the FOR UPDATE clause, INFORMIX-ESQL/C can usually perform the updates more quickly.

7. When you create a database with transactions that is not MODE ANSI, and do not use the BEGIN WORK and COMMIT WORK or ROLLBACK WORK statements, each UPDATE statement you execute is treated as a single transaction.
8. Each row affected by an UPDATE statement within a transaction is locked for the duration of the transaction; therefore, a single UPDATE statement that affects a large number of rows locks those rows until the entire operation is completed. If the number of rows affected is very large, you may exceed the limits placed by your operating system on the maximum number of simultaneous locks. If you exceed the limits, you may want to either reduce the number of rows affected by the UPDATE statement or lock the entire table before executing the statement.

See the section “Locking” in Chapter 1 for a more detailed description of table-level and row-level locking in INFORMIX-ESQL/C.

---

**Caution!** If INFORMIX-ESQL/C encounters an error while performing an UPDATE, the operation stops. Unless you have created the database with transactions, all database changes made up to the point where the error is encountered remain, but subsequent rows are not updated.

A data conversion error is an example of an error that stops an UPDATE operation. Common data conversion errors include attempting to insert numeric data into a CHAR column, or attempting to insert numeric values that exceed the limits of the data type of the column. For example, you cannot insert the integer 123456 into a SMALLINT column.

If you omit the WHERE clause, INFORMIX-ESQL/C assumes you want to update every row in the table.

---

## Examples

---

```
$ update stock
    set unit_price = unit_price * 1.04
    where manu_code = "HRO";

$ update stock set * =
    (5, "NRG", "tennis racquet", "295.00", "case",
     "10/case")
    where stock_num = 5 and manu_code = "NRG";

$ update customer
    set (fname, company, address2) =
        ("Marie", "Marie's Sports", "P. O. Box 3621")
    where customer_num = 103;

$ update table1
    set (col1, col2, col3) =
        ((select min (ship_charge), max (ship_charge)
          from orders), "07/01/1989")
    where col4 = 1001;
```

---

## Related Statements

**SELECT, DELETE, INSERT**



# UPDATE STATISTICS

## Overview

Use the `UPDATE STATISTICS` statement to cause the number of rows in a table to be recorded in the `systables` catalog.

## Syntax

---

`UPDATE STATISTICS [FOR TABLE table-name]`

---

## Explanation

<code>UPDATE STATISTICS</code>	are required keywords.
<code>FOR TABLE</code>	are optional keywords you use when you want to update the statistics for a single table.
<i>table-name</i>	is the name of the table for which you want the statistics updated.

## Notes

1. `UPDATE STATISTICS` is effective only when there is a current database.
2. `INFORMIX-ESQL/C` uses the data generated by `UPDATE STATISTICS` to optimize searching strategy. When you have modified a table extensively, use `UPDATE STATISTICS` to improve the efficiency of queries.
3. `INFORMIX-ESQL/C` does not update the statistics unless you execute the `UPDATE STATISTICS` statement.
4. When you do not use the `FOR TABLE` clause, `UPDATE STATISTICS` updates all the tables in the current database.

## Example

---

```
$ update statistics for table customer;
```

---

---

**INFORMIX-OnLine** supports additional functionality. Refer to the *INFORMIX-OnLine Programmer's Manual* for more information.

---

# WHENEVER

## Overview

Use the **WHENEVER** statement to trap errors that result during the execution of other statements.

## Syntax

---

```
WHENEVER {SQLERROR | NOT FOUND | SQLWARNING}  
        { GO TO label | GOTO label | CONTINUE }
```

---

## Explanation

**WHENEVER** is a required keyword.

**SQLERROR** is an optional keyword indicating that an error has occurred during an SQL statement. **ERROR** is a synonym that can be used instead of **SQLERROR**.

**NOT FOUND** are optional keywords indicating that you have attempted a **FETCH** beyond the last row in the active set or that there are no rows that satisfy the **SELECT** statement.

**SQLWARNING** is an optional keyword indicating a warning during an SQL statement.

**GO TO** are optional keywords.

**GOTO** is an optional keyword.

*label* is a statement label to which program control transfers when an error is encountered. If ANSI standard conformity is desired, you must prefix *label* with a colon ( : ).

**CONTINUE** is an optional keyword indicating that **INFORMIX-ESQL/C** should take no action. Use this option to turn off a previously set option.

## Notes

1. The **WHENEVER** statement is equivalent to placing an error-checking routine after every SQL statement.
2. The default for **SQLERROR** is **CONTINUE**. In the default situation, **INFORMIX-ESQL/C** does not test for errors.
3. The **WHENEVER SQLERROR** statement has modular scope, from the position of the **WHENEVER SQLERROR** statement to the next **WHENEVER SQLERROR** statement in the file (or to the end of the file if no more **WHENEVER SQLERROR** statements appear).

The scope of the **WHENEVER NOTFOUND** statement is the same.

4. You can include several **WHENEVER** statements in a program, and if they refer to the same keyword, the last one encountered at compile time takes precedence.
5. If you use **GO TO *label*** (or **GOTO *label***), every function that follows the **WHENEVER** statement must have a label with that name.
6. You receive a warning if you include the **WHENEVER SQLWARNING** keywords in a program and you have defined the **DBANSIWARN** environment variable or you compile with the **-ansi** flag.



## Examples

---

```
$ whenever sqlerror continue;
```

---

```
$ whenever sqlwarning goto progend;
```

```
...
```

```
progend:
```

```
printf("Program Over\n");
```

```
exit;
```

---

# The SELECT Statement

## Overview

Use the SELECT statement to query the current database.

The SELECT statement is made up of the following eight clauses. Only the SELECT clause and the FROM clause are required. If the INTO clause is present, it must precede the FROM clause.

## Syntax

---

```
SELECT clause [INTO clause] FROM clause  
              [WHERE clause]  
              [GROUP BY clause]  
              [HAVING clause]  
              [ORDER BY clause]  
              [INTO TEMP clause]
```

---

These clauses have the following syntax:

---

```
SELECT [ALL | DISTINCT] select-list
```

---

---

```
INTO variable-list
```

---

---

```
FROM {[OUTER] table-name [table-alias] | outer-expr} [...]
```

---

---

## WHERE *condition*

---

A *condition* is a collection of one or more *search conditions* connected by the logical operators AND, OR, or NOT. Examples of search conditions are as follows:

### 1. Comparison Condition

- a. *expr rel-op expr*
- b. *expr* [NOT] BETWEEN *expr* AND *expr*
- c. *expr* [NOT] IN (*value-list*)
- d. *column-name* [NOT] LIKE "*string*" [ESCAPE "*esc-char*"]
- e. *column-name* [NOT] MATCHES "*string*" [ESCAPE "*esc-char*"]
- f. *column-name* IS [NOT] NULL

### 2. Join Condition (a comparison condition among columns of the joined tables)

### 3. Condition with a Subquery

- a. *expr rel-op* {ALL | ANY | SOME} (*SELECT-statement*)
- b. *expr* [NOT] IN (*SELECT-statement*)
- c. [NOT] EXISTS (*SELECT-statement*)

---

## GROUP BY *column-list*

---

---

## HAVING *condition*

---

---

ORDER BY *column-name* [ASC | DESC][, ...]

---

---

INTO TEMP *table-name* [WITH NO LOG]

---

## Explanation

The following pages explain each of the syntax elements. A few basic concepts are defined here.

1. An *expression* consists of a column name, a host variable, or a constant, or any combination of these connected by the arithmetic operators:

Operator	Operation
+	addition
-	subtraction
*	multiplication
/	division

The result of the operation must make sense. For example, you cannot divide 16 by Jones.

Host variables in expressions must have a dollar sign (\$) or a colon (:) in front of them.

INFORMIX-ESQL/C has three functions that can be used wherever a constant can be used. TODAY always returns your system date. CURRENT returns the system date and time of day. USER returns a string containing the current user's login name.

An expression can also be one of the aggregate, date, or datetime functions. Do not include both an aggregate function and a column in an expression. The aggregate, date, and datetime functions that you can use in SQL statements are defined at the end of this chapter.



A CHAR column may have subscripts so that only a portion of the column value is involved in the expression. The notation for subscripting a column is *column-name*[*m*, *n*], where you want the *m*th through the *n*th characters of *column-name*. *m* must be less than or equal to *n*.

2. A *relational operator* is one of the following:

Operator	Operation
=	equal
!= or < >	not equal
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal

For CHAR expressions, “greater than” means “after” in ASCII collating order, where lowercase letters are after uppercase letters, and both are after numerals.

For DATE and DATETIME expressions, “greater than” means later in time; for INTERVAL expressions, “greater than” means of longer duration.

## Notes

1. The clauses of the SELECT statement are explained in detail on the following pages. Briefly, SELECT names a list of columns or expressions to be retrieved, INTO names the host variables to receive the data, FROM names a list of tables, WHERE sets conditions on the rows, GROUP BY groups rows together, HAVING sets conditions on the groups, ORDER BY orders the selected rows, and INTO TEMP puts the results into a temporary table.
2. When the SELECT statement returns no rows, INFORMIX-ESQL/C returns a “no rows found” code (`sqlca.sqlcode = SQLNOTFOUND`). See the section “Error Handling and the `sqlca` Structure” in Chapter 2 for a full explanation.

3. If a **SELECT** statement returns more than one row or if it is dynamically defined, you must use a cursor to **FETCH** one row at a time (see Chapter 1).
4. It is sometimes helpful to think of the **SELECT** statement as follows:

When you list more than one table in the **FROM** clause, **INFORMIX-ESQL/C** behaves as though it were creating a composite table that is the Cartesian product of all the tables in the **FROM** clause. That is, the rows of the new table are constructed by taking all the possible combinations of rows from all the tables listed in the **FROM** clause.

If there is a **WHERE** clause, **INFORMIX-ESQL/C** eliminates from this new table all rows that do not meet the conditions of the **WHERE** clause. This modified table is returned to the **SELECT** clause, where all columns not listed are eliminated. The resulting table is what the **SELECT** statement returns.

5. In a database created as **MODE ANSI**, the name of an object (table, index, view, synonym, and constraint) is qualified by the owner of the object (*owner.object*). You must specify *owner* when you refer to an object owned by another user.

The use of the prefix *owner.* is optional in a non-**MODE ANSI** database. **INFORMIX-ESQL/C** does check for the accuracy of *owner* if you include it in a statement, however. See the section “Owner Naming” in Chapter 1 for more information.

# ***SELECT Clause***

## **Overview**

Use the **SELECT** clause to specify the data you want to retrieve from one or more tables in a database.

## **Syntax**

---

```
SELECT [ALL | DISTINCT] select-list
```

---

## **Explanation**

**SELECT** is a required keyword.

**ALL** is a keyword that causes **INFORMIX-ESQL/C** to select all rows that satisfy the **WHERE** clause, without eliminating duplicates. This keyword is the default.

**DISTINCT** is a keyword that causes **INFORMIX-ESQL/C** to eliminate duplicate rows from the query results.

*select-list* is a list of column names and/or expressions separated by commas. A column name must be unambiguous; use its table name as a prefix to avoid confusion.

## **Notes**

1. When the **SELECT** statement does not include a **WHERE** clause, every row is returned.
2. The **DISTINCT** keyword can appear only once in each level of a query or subquery.
3. **UNIQUE** is a synonym for **DISTINCT**.



4. You can use the asterisk (\*) in the *select-list* to select all columns from all the tables in the FROM clause. You can produce the same result by listing every column name in the *select-list*.
5. To select all the columns from a single table, use the notation *tablename.\**.
6. You can supply a *display label* for the column name or an expression in the *select-list* by following the column name or expression with a legal identifier. The DESCRIBE statement enters the display label into the **sqlname** component of the **sqlvar\_struct** structure when it analyzes the SELECT statement. If you create a temporary table with the INTO TEMP clause, the column names of the temporary table are the display labels if they have been defined.
7. If you specify an aggregate function and a column in the *select-list*, the column must be used in the GROUP BY list (see GROUP BY for further explanation).
8. The SELECT clause cannot appear in a multi-statement PREPARE.

## Examples

The following examples use the INTO, FROM, and WHERE clauses, whose syntax is defined later.

---

```
$ select customer_num, lname, city
    into $cnum, $lname, $town
    from customer;
```

---

This statement selects columns **customer\_num**, **lname**, and **city**; the FROM clause indicates that these columns are taken from the **customer** table. The values returned are placed in the host variables **cnum**, **lname**, and **town**, respectively.



---

```
$ select count(*)  
    into $num  
    from orders  
    where customer_num = 101;
```

---

This statement counts the number of rows in **orders** in which the **customer\_num** column contains the value 101. In other words, it counts the number of orders made by the customer whose identifying number is 101. The number is placed in the variable **num**.

---

```
$ select avg(total_price)  
    from items  
    where order_num = 1021;
```

---

This statement computes the average of the **total\_price** values in those rows of **items** that contain an **order\_num** column equal to 1021.

---

```
$ select a+b abtotal, c*d cdprod  
    from tablez  
    into temp x;
```

---

This statement illustrates the use of display labels. It selects the sum of columns **a** and **b** from **tablez** and gives the sum the label **abtotal**. Similarly, the product of columns **c** and **d** is labeled **cdprod**. These labels are the values pointed to by the structure element **sqlvar\_struct.sqlname** associated with the **sqllda** structure that receives the results of the query.

# INTO Clause

Use the INTO clause to specify the host variables to receive the data that is retrieved by the SELECT statement.

## Syntax

---

INTO *variable-list*

---

## Explanation

INTO is a required keyword.

*variable-list* is a list of host variables (with or without attached indicator variables) that should agree in order and type with the corresponding columns or expressions in the *select-list*.

## Notes

1. You cannot use the INTO clause for dynamically created SELECT statements. You must use, instead, an **sqllda** structure to point to the data that is returned from the query.
2. If the SELECT statement stands alone (not in a DECLARE statement), it must be a singleton SELECT (returning exactly one row) and must have an INTO clause.
3. If the SELECT statement returns more than one row, you must use a cursor to FETCH the rows one at a time (see the section "Cursor Management" in Chapter 1). Put the INTO clause in the FETCH statement, rather than in the SELECT statement.

4. If the number of variables in *variable-list* differs from the number of items in the *select-list*, INFORMIX-ESQL/C returns a warning by setting `sqlca.sqlwarn.sqlwarn3` to W. The actual number of variables transferred is the lesser of the two numbers.
5. If possible, INFORMIX-ESQL/C converts the data type of each selected item to match that of the receiving variable. If the conversion is not possible, an error occurs and a negative value is returned in `sqlca.sqlcode`. In this case, the value in the host variable is unpredictable. See Chapter 4 for a discussion of data conversion.
6. You can use the optional INDICATOR keyword in the INTO clause of a SELECT statement. The use of the INDICATOR keyword conforms to ANSI standards. The indicator receives a number that warns the programmer of NULL values and truncated strings. You can use either a colon or the INDICATOR keyword to specify the indicator:

---

```
INTO variable[:indicator]  
INTO variable [INDICATOR indicator]
```

---

## Examples

---

```
$ declare q_curs cursor for  
    select lname, company  
    into $lname, $company  
    from customer;  
$ open q_curs;  
$ fetch q_curs;
```

---

or equivalently:

---

```
$  declare q_curs cursor for
      select lname, company
      from customer;
$  open q_curs;
$  fetch q_curs
      into $lname, $company;
```

---



# FROM Clause

## Overview

Use the FROM clause to specify the table or tables you want to select data from.

## Syntax

---

```
FROM {table-name [table-alias] | OUTER table-name [table-alias] |  
      OUTER (table-expr)} [, ...]
```

---

## Explanation

**FROM** is a required keyword.

**OUTER** is an optional keyword.

*table-name* is the name of a table that contains data for which you are searching.

*table-alias* is an optional alias for *table-name*.

*table-expr* is one or more of the options in the syntax shown above, for example ..., tab1, outer tab2 ... (See Appendix G, “Outer Joins,” for a discussion of this syntax.)

## Notes

1. Use the optional keyword OUTER to form outer joins. See the section “Join Condition” later in this chapter and Appendix G for a detailed description of outer joins.

2. You can supply an alias for a table name by following the table name with a space and an SQL identifier. This feature is especially useful when performing self-joins (see the section “WHERE Clause” later in this chapter).

## Examples

The following statement returns a row for every order:

---

```
$ select order_num, lname, fname
    from customer, orders
   where customer.customer_num
         = orders.customer_num;
```

---

In the following example, a row is returned for every order, as in the above statement. However, if a customer does not have an order, a row is still returned.

---

```
$ select order_num, lname, fname
    from customer, outer orders
   where customer.customer_num
         = orders.customer_num;
```

---

# WHERE Clause

## Overview

Use the WHERE clause to specify search criteria and join conditions on the data you want to select.

## Syntax

---

WHERE *condition*

---

## Explanation

WHERE is a required keyword.

*condition* is a collection of one or more *search conditions* connected by the logical operators AND, OR, or NOT. A search condition can be any of the following:

- Comparison condition
- Join condition
- Condition with a subquery

## Comparison Condition

A comparison condition can have one of the following forms:

- expr rel-op expr*
- expr* [NOT] BETWEEN *expr* AND *expr*
- expr* [NOT] IN (*value-list*)
- column-name* [NOT] LIKE "*string*" [ESCAPE "*esc-char*"]
- column-name* [NOT] MATCHES "*string*" [ESCAPE "*esc-char*"]
- column-name* IS [NOT] NULL

These forms are explained in the following pages. Any one of these conditions can be preceded by the keyword NOT, in which case, the negative of the condition must be satisfied.

Do not use an indicator variable in a WHERE clause to check for NULL values. INFORMIX-ESQL/C returns an error if you do. Use the IS [NOT] NULL option to check for NULL values.

## Syntax

---

*expr rel-op expr*

---

## Explanation

*expr* is an expression.

*rel-op* is a relational operator.

## Examples

---

```
$ select customer_num, order_date
    from orders
    where paid_date is null;

$ select fname, lname, company
    from customer
    where city[1,3] = "San";

$ select order_num, company
    from orders o, customer c
    where o.order_date > "6/12/89"
        and o.customer_num = c.customer_num;
```

---



## Syntax

---

*expr* [NOT] BETWEEN *expr* AND *expr*

---

### Explanation

<i>expr</i>	is an expression.
NOT	is a keyword that indicates that the expression to the left lies outside the range.
BETWEEN	is a keyword that indicates that the value of the expression to its left lies in the inclusive range of the values of the two expressions to its right.
AND	is a required keyword.

### Examples

---

```
$ select stock_num, manu_code
    from stock
   where unit_price between
        $lowprice and $hipprice;

$ select distinct customer_num, stock_num, manu_code
    from orders, items
   where order_date
        between "6/1/89" and "6/7/89";

$ select fname, lname
    from customer
   where zipcode not
        between "94100" and "94199";
```

---

## Syntax

---

*expr* [NOT] IN (*value-list*)

---

### Explanation

*expr* is an expression.

NOT is an optional keyword.

IN is a required keyword.

*value-list* is a list of values enclosed in parentheses.

### Notes

1. The search condition is satisfied when the expression to the left is included in the list of items. The NOT option produces a search condition that is satisfied when *expr* is not in the list of items.
2. *value-list* can contain host variables, constants, or the special keyword constants TODAY, CURRENT, and USER.
3. TODAY is evaluated at execution time. CURRENT is evaluated when a cursor is opened, or when the query is executed if you aren't using cursors.

### Examples

---

```
$ select lname, fname, company
   from customer
   where state in ("CA", "WA", "OR");

$ select item_num, total_price
   into $inum, $tprice
   from items
   where manu_code in ("HRO", "HSK");
```

---

## Syntax

---

*column-name* [NOT] LIKE "*string*" [ESCAPE "*escape-character*"]

---

### Explanation

<i>column-name</i>	is the name of a column.
NOT	is an optional keyword.
LIKE	is a required keyword.
<i>string</i>	is a pattern of characters enclosed in quotation marks.
ESCAPE	is an optional keyword.
<i>escape-character</i>	is a single character enclosed in quotation marks.

### Notes

1. The search condition is successful when the value of the column on the left matches the pattern specified by *string*. You can use wildcard characters in place of other characters in the string:  
  
    %     A percent sign matches zero or more characters.  
    \_     An underscore matches any single character.
2. The NOT option makes the search condition successful when the column on the left does not match the pattern specified by *string*.
3. The LIKE predicate in a WHERE clause allows an ESCAPE clause to specify an *escape-character*. You can use the ESCAPE keyword to specify an underscore ( `_` ) or the percent sign ( `%` ) in *string* and avoid their interpretation as wild card symbols.

For example, if *z* is the *escape-character*, then the characters *z\_* in a string evaluate to the character `_` (not the wild card). Similarly,

**z%** in the string evaluates to the character **%** (not the wild card). Finally, the characters **zz** in the string evaluate to the single character **z**.

4. You can only use the double quote ( **"** ) within *string* to match a literal quote; you cannot use the **ESCAPE** keyword. Similarly, you cannot use the quote character as the **ESCAPE** character in matching any other pattern.

## Examples

---

```
$ select fname, lname
   from customer
   where lname like "%son";
```

---

The previous example finds every customer representative whose last name ends in *son*.

---

```
$ select stock_num, manu_code, unit_price
   from stock
   where description like "%ball%";
```

---

The previous example obtains stock number, manufacturer code, and unit price for all stock items with *ball* anywhere in their description.

---

```
$ select * from customer
   where company like "%z_" escape "z";
```

---

The previous example retrieves all rows from the **customer** table where the value in the **company** column includes the underscore character.



## Syntax

---

*column-name* [NOT] MATCHES "*string*" [ESCAPE "*escape-character*"]

---

### Explanation

<i>column-name</i>	is the name of a column.
NOT	is an optional keyword.
MATCHES	is a required keyword.
<i>string</i>	is a pattern of characters enclosed in quotation marks.
ESCAPE	is an optional keyword.
<i>escape-character</i>	is a single character in quotation marks.

### Notes

1. The search condition is successful when the value of the column on the left matches the pattern specified by *string*. You can use wildcard characters in place of other characters in the string:

\* matches zero or more characters

? matches any single character

[...] matches any of the enclosed characters, including character ranges as in [a-z]. A caret (^) as the first character within the brackets matches any character that is *not* listed. [^abc] matches any character that is not a, b, or c.

\ escapes special significance of the next character (used to match \* or ? by writing \\* or \?)

2. The NOT option makes the search condition successful when the column on the left does not match the pattern specified by *string*.
3. The MATCHES comparison is implemented for compatibility with earlier versions of Informix products.
4. Values used in a MATCHES search must be type CHAR.
5. The MATCHES predicate in a WHERE clause allows an ESCAPE clause to specify an *escape-character*. You can use the ESCAPE keyword to specify the asterisk ( \* ), question mark ( ? ), left and right brackets ( [ , ] ), or backslash ( \ ) in *string* as characters (instead of as wild cards).

For example, if *z* is the *escape-character*, then *z?* evaluates to the character ? (not the wild card), and *zz* evaluates to the single character *z*.

6. You can only use the double quote ( " ) within *string* to match a literal quote; you cannot use the ESCAPE keyword. Similarly, you cannot use the quote character as the ESCAPE character in matching any other pattern.

## Examples

---

```
$ select fname, lname
   from customer
   where lname matches "Richard*";
```

---

The previous example selects rows in which the first seven letters of the last name are *Richard* (thus matching Richard, Richards, Richardson, and any others).

---

```
$ select customer_num, company
    from customer
    where city matches "[A-J]*";
```

---

The previous example provides the customer number and company name for all customers in cities that start with the letters *A* through *J*.

---

```
$ select * from customer
    where company matches "*z?*" escape "z";
```

---

The previous example retrieves rows from the **customer** table where the value in the **company** column includes a question mark.

## Syntax

---

*column-name* IS [NOT] NULL

---

## Explanation

*column-name* is the name of a column.

IS NULL are required keywords.

NOT is an optional keyword.

## Examples

---

```
$ select order_num, customer_num
    from orders
    where paid_date is null;
```

---

This statement lists those order numbers and customer numbers where the order has not been paid.

You can link any number of the above described conditions together using the logical operators AND or OR. For example,

---

```
$ select order_num, total_price
    from items
    where total_price > $loprice and
          manu_code like "H%";
```

```
$ select lname, customer_num
    from customer
    where zipcode between
          "93500" and "95700"
    or state not in
          ("CA", "WA", "OR");
```

---



# Join Condition

## Overview

You join two tables when you create a relationship in the **WHERE** clause between at least one column from one table and at least one column from the other. The effect of the join is to create a temporary composite table in which each pair of rows (one from each table) satisfying the join condition is linked together to form a single row.

## Syntax

The critical elements of a **SELECT** statement with a join between two tables *table1* and *table2* are as follows:

---

**SELECT** clause **FROM** *table1*, *table2* **WHERE** *condition*

---

## Explanation

<b>SELECT</b> clause	is any valid <b>SELECT</b> clause involving columns from <i>table1</i> or <i>table2</i> .
<b>FROM</b>	is a required keyword.
<i>table1</i>	is one of the tables in the join.
<i>table2</i>	is the other table in the join.
<b>WHERE</b>	is a required keyword.
<i>condition</i>	is any of the comparison conditions that involves columns from both <i>table1</i> and <i>table2</i> .

## Notes

1. When columns from different tables have the same name, you must distinguish them by prefixing the table identifier and a period, as in *table.column*.
2. A *multiple join* is a join of more than two tables. Its structure is similar to that shown previously, except that you have a join condition for more than one pair of tables in the FROM clause.
3. You can also join a table to itself in a *self-join*. To do so, you must list the table name twice in the FROM clause, assigning it two different aliases. Use the aliases to refer to each of the “two” tables in the WHERE clause.
4. An *outer join* occurs when every row from *table1* is taken, whether or not the join condition is met. If the join condition is not met, the columns from *table2* in the *select-list* are set to NULL values. You indicate an outer join by inserting the keyword OUTER before the table name *table2*.

## Examples

A two-table join:

---

```
$ select order_num, lname, fname
   from customer, orders
  where customer.customer_num
     = orders.customer_num;
```

---

This statement lists the order number and first and last names of the customer's representative for each order.

### A multiple-table join:

---

```
$ select distinct company, stock_num, manu_code
   from customer c, orders o, items i
  where c.customer_num = o.customer_num
        and o.order_num = i.order_num;
```

---

This statement yields the company name of the customer who ordered an item identified with the stock number and manufacturer code.

### A self-join:

---

```
$ select x.stock_num, x.manu_code,
        y.stock_num, y.manu_code
   from stock x, stock y
  where x.unit_price > 2.5 * y.unit_price;
```

---

This statement finds pairs of stock items whose unit prices differ by a factor greater than two and one-half. *x* and *y* are each aliases for the **stock** table.

### An outer join:

---

```
$ select company, order_num
   from customer c,
   outer orders o
  where c.customer_num = o.customer_num;
```

---

This statement lists the company name and all order numbers when the customer places an order. When no order is placed, the company name is still listed. See the section “Outer Joins” in Chapter 1 and Appendix G for information about outer joins.

# Condition with a Subquery

## Overview

The search condition in a `SELECT` statement can also

- Compare an expression to the result of another `SELECT` statement
- Determine whether an expression is included in the results of another `SELECT` statement
- Ask whether there are any rows selected by another `SELECT` statement

`SELECT` statements within a `WHERE` clause are called *subqueries*. A subquery can return a single value, no value, or a set of values, but it must have only a single column or expression in its *select-list* and must not contain an `ORDER BY` clause. You can make the subquery dependent upon the current row being evaluated by the outer `SELECT` statement (correlated subqueries).

## Syntax

---

`WHERE expr rel-op {ALL | [ANY | SOME]} (SELECT-statement)`

`WHERE expr [NOT] IN (SELECT-statement)`

`WHERE [NOT] EXISTS (SELECT-statement)`

---

## Explanation

`WHERE` is a required keyword.

*expr* is an expression.

*rel-op* is a relational operator.



ALL	is a keyword that denotes that the subquery can return zero, one, or more values and that the search condition is true if the comparison is true for each of the values returned. If the subquery returns no value, the search condition is true.
ANY	is a keyword that denotes that the subquery can return zero, one, or more values and that the search condition is true if the comparison is true for at least one of the values returned. If the subquery returns no value, the search condition is false.
SOME	is an alias for ANY.
NOT	is an optional keyword that reverses the truth value of the search condition.
IN	is a keyword that asks if <i>expr</i> is among the values returned by the following <i>SELECT-statement</i> .
EXISTS	is a keyword that asks if any rows are returned by the following <i>SELECT-statement</i> . The search condition is true if the subquery returns one or more rows.

*SELECT-statement* is a SELECT statement.

## Notes

1. The keywords ANY and ALL can be omitted in a comparison if you know the subquery will return exactly one value. In this case, the search condition is true if the comparison is true for the expression and the value returned by the subquery. The `sqlca.sqlcode` is set to a negative number when the subquery returns other than a single value.
2. *expr* IN (*SELECT-statement*) is equivalent to *expr* =ANY (*SELECT-statement*).

3. *expr* NOT IN (*SELECT-statement*) is equivalent to  
*expr* !=ALL (*SELECT-statement*).

## Examples

---

```
$ select order_num
   from items
  where stock_num = 9 and quantity =
        (select max(quantity) from items
         where stock_num = 9);
```

---

This statement returns a single value (the maximum number of volleyball nets ordered) to the outer-level query. The entire **SELECT** statement lists the order numbers for orders that include the maximum number of volleyball nets (**stock\_num = 9**).

---

```
$ select distinct order_num
   from items
  where total_price >all
        (select total_price
         from items
         where order_num = 1011);
```

---

This statement lists the order numbers of all orders containing an item whose total price is greater than the total price on all the items in order number 1011.

---

```
$ select distinct customer_num
   from orders
  where order_num not in
        (select order_num
         from items
         where stock_num = 1);
```

---

This statement lists all customer numbers corresponding to customers who have placed orders that do not include baseball gloves (`stock_num = 1`).

---

```
$ select order_num, stock_num, manu_code, total_price
   from items x
   where total_price >
      (select 2 * min(total_price)
       from items
       where order_num = x.order_num);
```

---

This statement (using a correlated subquery) lists all items whose total price is at least twice the minimum total price for all items on the same order.

# GROUP BY Clause

## Overview

Use the GROUP BY clause to produce a single row of results for each group. A group is a set of rows having the same values for each column listed.

## Syntax

---

GROUP BY *group-list*

---

## Explanation

GROUP BY      are required keywords.

*group-list*      is a column name or a list of column names, separated by commas, that determines the group. The query result contains a single row for each set of rows that satisfies the WHERE clause and contains a unique value or set of values in the column or columns indicated by *group-list*.

## Notes

1. Using a GROUP BY clause restricts what you can enter in the SELECT clause. The *select-list* can include aggregate functions for any column and/or the name of any column that you also list in the GROUP BY clause. The *select-list* can also contain an expression that involves columns in *group-list*. Do not, however, list any column in *select-list* that you do not also list in *group-list*.
2. NULL values are considered identical (as a single group) when evaluated within a GROUP BY clause.



3. In the place of column names in *group-list*, you can enter one or more integers that refer to the position of items in the *select-list*.

## Examples

---

```
$ select order_num, count(*), sum(total_price)
    from items
    group by order_num;
```

---

The previous statement obtains the number of items and total price of all items for each order.

---

```
$ select order_num, count(*), sum(total_price)
    from items
    group by 1;
```

---

The previous statement returns the same information as the example before.

---

```
$ select order_date, paid_date, paid_date - order_date
    from orders
    group by 1,2;
```

---

The previous statement is an example of a *select-list* that contains an expression involving columns in the *group-list*.

# HAVING Clause

## Overview

Use the HAVING clause to apply one or more qualifying conditions to groups.

## Syntax

---

HAVING *condition*

---

## Explanation

HAVING is a required keyword.

*condition* is a condition as defined for the WHERE clause.

## Notes

1. Each condition compares one aggregate property of the group either with another aggregate property of the group or with a constant.
2. The HAVING clause generally complements a GROUP BY clause. If you use HAVING without GROUP BY, the HAVING clause applies to all rows that satisfy the WHERE clause. Without a GROUP BY clause, all rows that satisfy the WHERE clause make up a single group.
3. You can use the HAVING clause to place conditions on the GROUP BY column values as well as on aggregate values.

## Examples

---

```
$ select order_num, avg(total_price)
   from items
   group by order_num
   having count(*) > 2;
```

---

This statement returns average total price per item on all orders that have more than two items.

# ORDER BY Clause

## Overview

Use the ORDER BY clause to sort query results by the values contained in one or more columns. You can sort only by a column that you select in the SELECT clause.

## Syntax

---

ORDER BY *column-name* [ASC | DESC][, ...]

---

## Explanation

ORDER BY      are required keywords.

*column-name*      is the name of a column by which you want to sort the query results.

ASC              is a keyword that specifies that the results are in ascending order. ASC is the default.

DESC              is a keyword that specifies that the results are in descending order.

## Notes

1. You can ORDER BY up to eight columns.
2. You can only ORDER BY columns that are named explicitly or implicitly in the *select-list*.
3. The total length of the data in the columns included in the ORDER BY clause cannot be greater than 120 bytes.



4. In the place of column names, you can enter one or more integers that refer to the position of items in the *select-list*. In this way, you can ORDER BY an expression.

## Examples

---

```
CORRECT:  $  select order_date, ship_date
              from orders
              order by order_date;
```

```
CORRECT:  $  select *
              from orders
              order by order_date;
```

```
INCORRECT: $  select order_date, ship_date
              from orders
              order by customer_num;
```

---

In the first two statements, **order\_date** is either explicitly or implicitly listed in the *select-list*. In the third statement, even though **customer\_num** is in the **orders** table, it is not in the *select-list*.

---

```
$  select customer_num, fname, lname, company
       from customer
       order by 3, 2;
```

---

This statement performs a nested sort: the first level is the **lname** column; the second level is the **fname** column.

# INTO TEMP Clause

## Overview

Use the INTO TEMP clause to create a temporary table that contains the query results. The temporary table disappears when your program ends. When a database uses logging, you can prevent logging of a temporary table by using the WITH NO LOG option.

## Syntax

---

INTO TEMP *table-name* [WITH NO LOG]

---

## Explanation

INTO TEMP      are required keywords.

*table-name*      is the SQL identifier you want to assign to the temporary table.

WITH NO LOG      are optional keywords to prevent logging of temporary tables in a database that uses logging.

## Notes

1. You save time if you use a temporary table when the same query results are required several times.
2. An INTO TEMP clause in a SELECT statement often gives you clearer and more easily understood statements.
3. The column names of the temporary table are those named in the *select-list*. If you list a display label for a column or expression, the column name in the temporary table is the display label.
4. There are no indexes associated with *table-name*.

5. It is an error to use an INTO clause with the INTO TEMP clause. When you do, no results are returned to the host variables, and **sqlca.sqlcode** is set to a negative value.
6. The temporary table persists until you exit your **INFORMIX-ESQL/C** program or issue the **DROP TABLE** statement for it.
7. If you use the **WITH NO LOG** keywords in a **SELECT INTO TEMP** statement and the database does not use logging, the **WITH NO LOG** option is ignored. The **WITH NO LOG** option is supported on a **MODE ANSI** database.
8. Once you turn off the logging on a temporary table, you cannot turn it back on again; a temporary table is therefore always logged or never logged.

# UNION Operator

## Overview

The UNION operator is a keyword placed between two SELECT statements that let you combine the queries into a single query.

## Syntax

---

```
SELECT-statement UNION [ALL] SELECT-statement  
[UNION [ALL] SELECT-statement ...]
```

---

## Explanation

*SELECT-statement* is a SELECT statement.

UNION is a keyword that selects all rows from both queries, removes duplicates, and returns what is left.

ALL is an optional keyword that leaves the duplicates.

## Notes

1. The UNION operator can be placed between each member of a sequence of more than two queries.
2. It is possible to write single queries that are equivalent to the compound queries constructed using the UNION operator. The advantage of the UNION operator is that it makes the process easier.
3. There are restrictions on the queries that you can connect with UNION operators:



- The number of the items in the *select-list* of each query must be the same, and corresponding items in each *select-list* must have identical data types.
  - Corresponding items need not have the same identifier.
  - Do not use an INTO statement in a query unless you are sure that the compound query will return exactly one row, and you are not using a cursor. In the case where you must do this when these conditions do not prevail, be sure that the INTO clause is in the first SELECT statement.
  - If you use an ORDER BY clause, it must follow the last SELECT statement, and you must refer by integer, not by identifier, to the item to be ordered. Ordering takes place after the set operation is complete.
4. UNION operators must not occur inside a subquery and must not be used in the definition of a view.
  5. The column names (or display labels) of the resulting table are the same as those from the first SELECT statement.
  6. You can put the results of a UNION into a temporary table by putting an INTO TEMP clause in the final SELECT statement.

## Example

The following statement selects those items (identified by a stock number and a manufacturer code) that have a unit price of less than \$100.00 or that have been ordered in quantities greater than three.

---

```
$  select distinct stock_num, manu_code
    from stock
    where unit_price < 100.00

union

select stock_num, manu_code
    from items
    where quantity > 3
    order by 1;
```

---

## ***Functions in SQL Statements***

You can use the aggregate, length, date, and datetime functions anywhere in an SQL expression that a constant appears. The following functions are available:

<b>Aggregate Functions</b>	<b>Length Function</b>	<b>Date Functions</b>	<b>Datetime Functions</b>
COUNT( ) SUM( ) AVG( ) MAX( ) MIN( )	LENGTH( )	DATE( ) DAY( ) MDY( ) MONTH( ) WEEKDAY( ) YEAR( )	CURRENT EXTEND( )

These functions are defined on the pages that follow.

You can also use the TODAY and USER functions that are described in Chapter 1 of this manual.

# Aggregate Functions

## Overview

The aggregate functions take on values that depend on the set of rows returned by the WHERE clause of a SELECT statement. In the absence of a WHERE clause, the aggregate functions take on values that depend on all the rows formed by the FROM clause.

Syntax	Function
COUNT(*)	returns the number of rows that satisfy the WHERE clause.
COUNT(DISTINCT <i>x</i> )	returns the number of unique values in column <i>x</i> that satisfy the WHERE clause.
SUM( <u>ALL</u>   DISTINCT] <i>x</i> )	returns the sum of all values in column <i>x</i> that satisfy the WHERE clause. You can apply SUM only to number columns. If you use the keyword DISTINCT, the sum is only over distinct values in column <i>x</i> . ALL is the default.
AVG( <u>ALL</u>   DISTINCT] <i>x</i> )	returns the average of all values in column <i>x</i> that satisfy the WHERE clause. You can apply AVG only to number columns. If you use the keyword DISTINCT, the average is only over distinct values in column <i>x</i> . ALL is the default.
MAX( <u>ALL</u>   DISTINCT] <i>x</i> )	returns the highest value contained in column <i>x</i> from a row that satisfies the WHERE clause. If you use the keyword DISTINCT, the maximum is only over distinct values in column <i>x</i> . ALL is the default.



**MIN([ALL | DISTINCT] *x*)**

returns the lowest value contained in column *x* from a row that satisfies the WHERE clause. If you use the keyword DISTINCT, the minimum is only over distinct values in column *x*. ALL is the default.

## Notes

1. In the functions SUM(*x*), AVG(*x*), MAX(*x*), and MIN(*x*), *x* can be an expression instead of a column. In this case, the function is evaluated over the values of the expression as computed for each row that satisfies the WHERE clause. Do not include another aggregate function in the expression.
2. Use the DISTINCT keyword only with columns, not with expressions.
3. You can use the keyword DISTINCT only once in each level of a query or subquery. Use DISTINCT to eliminate duplicate query results or to eliminate duplicates from the argument of an aggregate function.
4. NULL values affect the aggregate functions in the following ways:
  - COUNT(\*) counts all rows, even if the value of every column in the row is NULL.
  - COUNT(DISTINCT *x*), AVG (*x*), SUM (*x*), MAX (*x*), and MIN (*x*) ignore rows with NULL values for *x* and return the appropriate values based on the rest of the rows.
  - If *x* contains only NULL values, then COUNT(DISTINCT *x*) returns zero, and the other four aggregate functions return NULL for that column.
5. UNIQUE is a synonym for DISTINCT.

# LENGTH( )

## Overview

The LENGTH function returns the number of bytes remaining in the referenced column or string after deleting any trailing spaces.

## Syntax

---

LENGTH ( *string* )

---

## Explanation

*string* is a CHAR variable or a string constant.

## Note

LENGTH allows only one argument value. For example, you cannot determine the total number of characters in a customer's name by entering length(fname,lname). However, you can combine LENGTHs through an expression.

## Example

---

```
$ select customer_num, length(fname) + length(lname),  
      length("How many bytes is this?")  
      from customer  
      where length(company) > 10;
```

---

---

INFORMIX-OnLine supports additional data types. Refer to the *INFORMIX-OnLine Programmer's Manual* for more information.

---

# DATE( )

## Overview

The DATE function returns a type DATE value corresponding to the expression with which you call it.

## Syntax

---

DATE(*expr*)

---

## Explanation

DATE is a required keyword.

*expr* is a required expression that can be converted to a type DATE value.

## Example

The following statement uses DATE to convert a string to a date:

---

```
where end_date > date("12/13/1989")
```

---

date(100) returns the 100<sup>th</sup> day after December 31, 1899.

# DAY( )

## Overview

The DAY function returns the day of the month when you call it with a type DATE or DATETIME expression.

## Syntax

---

**DAY**(*date-expr*)

---

## Explanation

**DAY** is a required keyword.

*date-expr* is a required expression of type DATE or DATETIME.



# MDY( )

## Overview

The MDY function returns a type DATE value when you call it with three expressions that evaluate to integers representing the month, date, and year.

## Syntax

---

MDY(*expr1*, *expr2*, *expr3*)

---

## Explanation

MDY	is a required keyword.
<i>expr1</i>	is an expression that evaluates to an integer representing the number of the month (1-12).
<i>expr2</i>	is an expression that evaluates to an integer representing the number of the day of the month (1-28, 29, 30, or 31, depending on the month).
<i>expr3</i>	is an expression that evaluates to a four-digit integer representing the year.

## Note

The value of *expr3* cannot be the abbreviation for the year because any two-digit integer is interpreted as in the first century.

# MONTH( )

## Overview

The MONTH function returns an integer corresponding to its type DATE or DATETIME argument.

## Syntax

---

MONTH(*date-expr*)

---

## Explanation

**MONTH** is a required keyword.

*date-expr* is a required expression of type DATE or DATETIME.

# WEEKDAY( )

## Overview

The WEEKDAY function returns an integer that represents the day of the week when you call it with a type DATE or DATETIME expression.

## Syntax

---

WEEKDAY(*date-expr*)

---

## Explanation

**WEEKDAY** is a required keyword.

*date-expr* is a required expression of type DATE or DATETIME.

## Note

WEEKDAY returns an integer in the range 0-6; 0 represents Sunday, 1 represents Monday, and so on.

# YEAR( )

## Overview

The YEAR function returns an integer that represents the year when you call it with a type DATE or DATETIME expression.

## Syntax

---

YEAR(*date-expr*)

---

## Explanation

**YEAR** is a required keyword.

*date-expr* is a required expression of type DATE or DATETIME.



# CURRENT

## Overview

The **CURRENT** function returns a **DATETIME** value with today's current date and time.

## Syntax

---

**CURRENT** [*first* **TO** *last*]

---

## Explanation

**CURRENT** is a required keyword.

*first* is an optional qualifier that specifies the first field to be returned.

**TO** is a required keyword if you include *first* and *last* qualifiers.

*last* is an optional qualifier that specifies the last field to be returned.

## Notes

1. The value returned by the **CURRENT** function is the date and time it is executed.
2. You can use the **CURRENT** function in any context in which you can use a **DATETIME** literal.
3. The *first* qualifier must specify a field that is more significant than or equal to the *last* qualifier.

4. If the function is executed more than once in a statement, identical values may be returned at each point of the call. You cannot rely on the **CURRENT** function to provide distinct values each time it is executed.
5. **INFORMIX-ESQL/C** may not execute the **CURRENT** function in the physical order in which it appears in a statement. You should not use the function to mark the start, the end, or a particular point in the execution of the statement.
6. If no *first* TO *last* qualifiers are specified, the default qualifiers are **YEAR TO FRACTION**.
7. The following are valid qualifiers:

Identifier	Qualified Data
------------	----------------

<b>YEAR</b>	A number of years.
-------------	--------------------

<b>MONTH</b>	A number of months.
--------------	---------------------

<b>DAY</b>	A number of days.
------------	-------------------

<b>HOURL</b>	A number of hours.
--------------	--------------------

<b>MINUTE</b>	A number of minutes.
---------------	----------------------

<b>SECOND</b>	A number of seconds.
---------------	----------------------

<b>FRACTION</b>	A decimal fraction of a second, with up to 5 digits of precision. The default precision is 3 digits (thousandths of a second).
-----------------	--

### Example

---

```
$ select prog_title from tv_programs where  
    air_date > current year to day;
```

---

# EXTEND( )

## Overview

The EXTEND function adjusts the precision of a DATETIME value.

## Syntax

---

EXTEND(*value*[,*first* TO *last*])

---

## Explanation

**EXTEND** is a required keyword.

*value* is a DATETIME or DATE type column name, variable, or expression.

*first* is an optional qualifier that specifies the first field in the result.

**TO** is a required keyword if you include *first* and *last* qualifiers.

*last* is an optional qualifier that specifies the last field in the result.

## Notes

1. The *value* can also be a DATETIME literal or a character string in a valid DATETIME format. (It cannot be the string representation of a DATE value.)
2. If no *first* TO *last* qualifiers are specified, the default qualifiers are YEAR TO FRACTION.

3. The *first* qualifier must specify a field that is more significant than or equal to the *last* qualifier.
4. If the *value* contains fields not specified by the qualifiers, the unwanted fields are discarded.
5. If the *first* qualifier specifies a field to the left of (that is, more significant than) what exists in *value*, the new fields are filled with values returned by the CURRENT function.
6. If the *last* qualifier specifies a field to the right of (that is, less significant than) what exists in *value*, the new fields are filled in with constant values. A missing MONTH or DAY is filled in with 1, and the missing fields HOUR TO FRACTION are filled in with 0.
7. The following are valid qualifiers:

Identifier	Qualified Data
------------	----------------

YEAR	A number of years.
------	--------------------

MONTH	A number of months.
-------	---------------------

DAY	A number of days.
-----	-------------------

HOUR	A number of hours.
------	--------------------

MINUTE	A number of minutes.
--------	----------------------

SECOND	A number of seconds.
--------	----------------------

FRACTION	A decimal fraction of a second with up to 5 digits of precision. The default precision is 3 digits (thousandths of a second).
----------	---

8. If an INTERVAL value includes a field that is not present in a DATETIME value, you cannot combine the two values with the addition ( + ) or subtraction ( - ) operators. Similarly, you cannot add or subtract a DATE value and an INTERVAL whose *last* qualifier is smaller than DAY.

In these cases, you must use the EXTEND function to return an adjusted DATETIME value on which to perform the arithmetic operation.



## Examples

In the following example, the EXTEND function returns the MONTH and DAY fields from a column that contains YEAR, MONTH, and DAY data. In this instance, the YEAR field is not returned.

---

```
$ select extend(air_date, month to day)
   from tv_programs;
```

---

In this example, assume that `air_date` is a DATE column, or else a DATETIME column whose *last* qualifier is larger than MINUTE. Then the EXTEND function is required in the following statement, because the INTERVAL operand includes a field (MINUTE) that is not part of the precision of `air_date`.

---

```
$ select sponsor, prog_title, air_date from tv_programs
   where air_date >= today
      and current > extend(air_date) - 2880 units minute;
```

---

Since no qualifiers are specified as arguments, the EXTEND function returns a DATETIME value with the default precision YEAR to FRACTION(3). The WHERE clause in this example specifies the rows in which `air_date` has a value in a range from the beginning of the current day (TODAY) up to 48 hours (= 2,880 minutes) after the current instant.

See also Appendix H, "Working with DATETIME and INTERVAL Data," for more information about arithmetic operations on date-time data types.

# **Chapter 4**

## **SQL Data Types**



## Chapter 4 Table of Contents

Chapter Overview .....	5
Correspondence Between SQL and C .....	6
Data Conversion .....	8
CHAR Type.....	10
SMALLINT and INTEGER Types.....	11
SERIAL Type.....	11
SMALLFLOAT and FLOAT Types .....	12
DATE Type .....	12
RDATESTR .....	14
RDAYOFWEEK .....	15
RDEFMTDATE .....	16
RFMTDATE .....	18
RJULMDY .....	21
RLEAPYEAR .....	22
RMDYJUL .....	23
RSTRDATE .....	24
RTODAY .....	25
MONEY Type .....	26
DECIMAL Type .....	26
DECIMAL Type Routines .....	28
DECCVASC.....	29
DECTOASC .....	31
DECCVINT .....	33
DECTOINT .....	34
DECCVLONG .....	35
DECTOLONG .....	36
DECCVDBL .....	37
DECTODBL .....	38
DECADD, DECSUB, DECMUL, and DECDIV .....	39
DECCMP .....	41
DECCOPY .....	42
DECECVT and DECFCVT .....	43
DECROUND .....	45
DECTRUNC.....	46



Numeric Formatting Routines .....	47
RFMTDEC .....	48
RFMTDOUBLE .....	49
RFMTLONG .....	50
Formatting Numeric Strings .....	51
 DATETIME and INTERVAL Type Routines.....	57
DTCURRENT .....	58
DTCVASC .....	60
DTEXTEND .....	62
DTTOASC .....	64
INCVASC .....	66
INTOASC .....	68
DATETIME and INTERVAL Types .....	70
DATETIME Columns.....	71
DATETIME <i>first TO last</i> .....	72
INTERVAL Columns .....	73
INTERVAL <i>first TO last</i> .....	74
FETCHing DATETIME and INTERVAL Values .....	75
FETCH DATETIME Values .....	75
Fetching INTERVAL Values .....	75
Data Conversion When FETCHing .....	76
Storing DATETIME and INTERVAL Values.....	76
Data Conversion When Storing.....	76
Declaration of DATETIME and INTERVAL.....	77
Converting Between DATETIME and DATE.....	78

## Chapter Overview

This chapter describes the data types used in **INFORMIX-ESQL/C** for columns in a database and shows how the data types are represented in the C language. It includes library functions for data conversion and manipulation for the **DATE**, **DECIMAL**, **DATETIME**, and **INTERVAL** data types and numeric formatting routines.

## Correspondence Between SQL and C

The following example lists the SQL data types and their C language counterparts. Use these to select the appropriate C data type for host variables corresponding to columns in the database. Valid synonyms for the data types are listed in Chapter 1.

---

SQL Data Type	C Data Type
CHAR(n)	char array[n+1] string array[n+1] char * fixchar array[n]
SMALLINT	short int
INTEGER	long int
SERIAL	long int
SMALLFLOAT	float
FLOAT	double
DATE	long int
MONEY	dec_t or struct decimal
DECIMAL	dec_t or struct decimal
DATETIME	dtime_t or struct dtime
INTERVAL	intrvl_t or struct intrvl

---

The **string** and **fixchar** C data types are defined later in the section on the **CHAR** data type.

See the section in this chapter on the **DECIMAL** data type for a full explanation of the **DECIMAL** data type and the **dec\_t** structure.

Within a C program using dynamically defined SQL statements, you can identify the data type returned to an **sqli** structure by the **DESCRIBE** statement, by using the following constants defined in **sqltypes.h**:

---

Constant	SQL Data Type
SQLCHAR	CHAR
SQLSMINT	SMALLINT
SQLINT	INTEGER
SQLFLOAT	FLOAT
SQLSMFLOAT	SMALLFLOAT
SQLDECIMAL	DECIMAL
SQLSERIAL	SERIAL
SQLDATE	DATE
SQLMONEY	MONEY
SQLDTIME	DATETIME
SQLINTERVAL	INTERVAL

---

When you fill an `sqlda` structure to indicate where and what kind of host variables are used as input parameters to a dynamically defined SQL statement, you can use the following constants (also defined in `sqltypes.h`) to identify the C data type.

---

Constant	C Data Type
CCHARTYPE	char
CSHORTTYPE	short int
CINTTYPE	int
CLONGTYPE	long
CFLOATTYPE	float
CDOUBLETTYPE	double
CDECIMALTYPE	dec_t or struct decimal
CFIXCHARTYPE	fixchar
CSTRINGTYPE	string
CDATETIME	dtime_t or struct dtime
CINTERVAL	interval_t or struct intrvl

---



---

**INFORMIX-OnLine** supports additional data types. Refer to the *INFORMIX-OnLine Programmer's Manual* for more information.

---



## Data Conversion

When there is a discrepancy between the data type of a database variable and that of the host variable, or between the data type of two columns, **INFORMIX-ESQL/C** attempts to convert one into the other. This includes the conversion of a **CHAR** type into a number type when the **CHAR** variable is a representation of a number. When a comparison is made between a **CHAR** value and a number value, for example, **INFORMIX-ESQL/C** converts the **CHAR** value to a number value.

The conversion of a number type to a character type occurs through the creation of a string. **INFORMIX-ESQL/C** uses an exponential format for very large or very small numbers.

If conversion is not possible, either because it makes no sense or because the receiving variable is too small to accept the converted value, **INFORMIX-ESQL/C** returns values as described in the following table, where "N" represents a number type and "C" a character type.

---

Conversion	Problem	Result
C → C	does not fit	String is truncated; <b>sqlca.sqlwarn.sqlwarn1</b> is set to W; indicator variable set to size of original string.
N → C	does not fit	String is filled with asterisks; <b>sqlca.sqlwarn.sqlwarn1</b> is set to W; indicator variable is set to positive integer.
C → N	not number	Number is undefined; <b>sqlca.sqlcode</b> is set to negative.
C → N	overflow	Number is undefined; <b>sqlca.sqlcode</b> is negative.
N → N	overflow	Number is undefined; <b>sqlca.sqlcode</b> is negative.

---

**INFORMIX-ESQL/C** carries out all arithmetic in an arithmetic expression in type **DECIMAL**. The type of the resulting variable determines the

format of the stored or printed result. The following rules apply to the precision and scale of the DECIMAL variable that results from an arithmetic operation on two numbers:

- All operands, if not already DECIMAL, are converted to DECIMAL, and the resulting number is DECIMAL.

Convert Type	To
FLOAT	DECIMAL(16)
SMALLFLOAT	DECIMAL(8)
INTEGER	DECIMAL(10,0)
SMALLINT	DECIMAL(5,0)

- The precision and scale of the result of an arithmetic operation depend on the precision and scale of the operands and on the type of arithmetic expression. The rules are summarized in the example at the end of this section for arithmetic operations on operands with definite scale. When one of the operands has no scale (floating decimal), the result is a floating decimal.
- If the operation is addition or subtraction, INFORMIX-ESQL/C adds trailing zeros to the operand with the smallest scale until the scales are equal.
- If the type of the result of an arithmetic operation requires the loss of significant digits, INFORMIX-ESQL/C reports an error.
- Leading or trailing zeros are not considered significant digits and do not contribute to the determination of precision and scale.

In the following table, let  $p_1$  and  $s_1$  be the precision and scale of the first operand, and let  $p_2$  and  $s_2$  be the precision and scale of the second operand.

---

Operation	Precision and Scale of Result
Addition and Subtraction	Precision: $\text{MIN}(32, \text{MAX}(p_1 - s_1, p_2 - s_2) + \text{MAX}(s_1, s_2) + 1)$ Scale: $\text{MAX}(s_1, s_2)$
Multiplication	Precision: $\text{MIN}(32, p_1 + p_2)$ Scale: $s_1 + s_2$
Division	Precision: 32 Scale: $32 - p_1 + s_1 - s_2$ (cannot be negative)

---



---

INFORMIX-OnLine supports additional data types. Refer to the *INFORMIX-OnLine Programmer's Manual* for more information.

---

## CHAR Type

The CHAR data type is a character string whose length can vary from 1 to 32,511. When you associate a column with a CHAR data type in the CREATE TABLE statement, you must indicate its length with the notation CHAR(*n*). Within the data file, CHAR(*n*) is stored in *n* bytes. Because character strings in C are usually terminated with a null byte, you should declare the host array that receives a CHAR(*n*) value with a size of *n* + 1.

If you choose to declare a host variable as a pointer to a char, be sure to allocate sufficient memory to receive the longest CHAR value. If you do not, you may overwrite other data when you transfer data from the database to your host variable.

The INFORMIX-ESQL/C preprocessor also recognizes the **string** and **fixchar** C data types for host variables.

- The **char** data type pads with trailing blanks up to the size of the CHAR column returned. It is null-terminated.



- The **string** data type differs from the **char** data type by truncating trailing blanks before inserting the null character to signal the end of the string.
- The **fixchar** data type is the same as the **char** except that it does not add the trailing null byte to terminate the string. This means that you can declare a **fixchar** host variable corresponding to a **CHAR(*n*)** column as an array with *n* components.

## SMALLINT and INTEGER Types

**SMALLINT** and **INTEGER** are SQL data types that correspond to the C data types **short** and **long**. **SMALLINT** values can range from -32,767 to +32,767, while **INTEGER** values can range from -2,147,483,647 to +2,147,483,647. The values -32,768 for **SMALLINT** and -2,147,483,648 for **INTEGER** are reserved and cannot be used. Within a data file, **SMALLINT** requires two bytes and **INTEGER** four bytes.

## SERIAL Type

A **SERIAL** data type is a sequential integer assigned automatically by **INFORMIX-ESQL/C** when you execute an **INSERT** statement (see Chapter 2). The **SERIAL** data type corresponds to a **long C** type and occupies four bytes in a data file. When creating a table, you can indicate the starting number *n* ( $\geq 1$ ) for a **SERIAL** data type by declaring a column as **SERIAL(*n*)**.



## SMALLFLOAT and FLOAT Types

The SMALLFLOAT and FLOAT data types are implemented as **float** and **double** C data types, respectively. This means that they are binary floating-point numbers. You can experience round-off problems when they are expressed in decimal notation. You may prefer to use the DECIMAL data type in place of SMALLFLOAT and FLOAT to avoid round-off.

You can convert DECIMAL columns to float and double host variables to take advantage of the C mathematics library. You also can use float and double type host variables as input parameters in SQL statements.

Since the range of allowed DECIMAL values is much larger than the range allowed to floats or doubles, you should check that the magnitude of the DECIMAL value is less than the maximum magnitude for floats and doubles on your machine before making the conversion from DECIMAL to float or double. You can find functions that convert in both directions in the section on the DECIMAL data types.

## DATE Type

INFORMIX-ESQL/C stores dates as a four-byte integer whose value is the number of days since December 31, 1899. Dates before December 31, 1899, are negative numbers while dates after December 31, 1899, are positive numbers. You can add and subtract numbers from DATE type values to produce a value corresponding to a date that many days later or earlier. You also can subtract two DATE types to get the number of days between them.

The following date manipulation routines are included in the libraries distributed with INFORMIX-ESQL/C for converting dates written in string form to and from this internal format. These are described on the next several pages.

---

<b>rdatestr</b>	Convert internal format to string
<b>rdayofweek</b>	Return day of week
<b>rdefmtdate</b>	Convert string to internal format
<b>rfmtdate</b>	Convert internal format to string
<b>rjulmdy</b>	Return month, day, and year from internal format
<b>rleapyear</b>	Determine whether leap year
<b>rmdyjul</b>	Return internal format from month, day, and year
<b>rstrdate</b>	Convert string to internal format
<b>rtoday</b>	Return system date in internal format

---

# RDATESTR

## Overview

Use `rdatestr` to convert a date in internal format to a character string date of the form *mm/dd/yyyy*.

## Syntax

---

```
rdatestr(jdate, str)  
    long jdate;  
    char *str;
```

---

## Explanation

*jdate* is the internal representation of a date as a long integer.

*str* is a pointer to the area where the results are to be stored.

# RDAYOFWEEK

## Overview

The **rdayofweek** routine returns the day of the week represented as an integer, given an internal date as an argument.

## Syntax

---

```
rdayofweek(jdate)  
    long jdate;
```

---

## Explanation

*jdate* is the internal representation of the date as a long integer.

## Return Values

- 0 Sunday
- 1 Monday
- 2 Tuesday
- 3 Wednesday
- 4 Thursday
- 5 Friday
- 6 Saturday



# RDEFMTDATE

## Overview

Use **rdefmtdate** to create a long integer whose value is the number of days since December 31, 1899, for a string date whose format is provided.

## Syntax

---

```
rdefmtdate(jdate, fmtstring, input)
    long *jdate;
    char *fmtstring;
    char *input;
```

---

## Explanation

- jdate* is a long integer, the internal representation of the date expressed as *input*.
- fmtstring* is a pointer to a character array containing the format pattern for the date supplied in *input*.
- input* is a pointer to the string containing the date to be converted to a long integer.

## Notes

1. The string *fmtstring* uses the same formatting characters as **rfmtdate**.
2. The *input* string and the *fmtstring* must be in the same sequential order in terms of month, day, and year. They need not, however, have the same literals nor the same representation for month, day, and year.

## Return Codes

- 0      Operation was successful.
- 1204      There is an invalid year component in the *input* parameter.
- 1205      There is an invalid month component in the *input* parameter.
- 1206      There is an invalid day component in the *input* parameter.
- 1209      Since *\*input* does not contain delimiters between the year, month, and day components, the length of *\*input* must be exactly six or eight bytes.
- 1212      *fmtstring* does not contain a year, a month, and a day component.

## Examples

The following are valid combinations of *fmtstring* and *input*:

---

<b>fmtstring</b>	<b>input</b>
"mmdyy"	"Dec. 25th, 1989"
"mmm. dd. yyyy"	"dec 25 1989"
"mmm. dd. yyyy"	"DEC-25-1989"
"mmm. dd. yyyy"	"122589"
"mmm. dd. yyyy"	"12/25/89"
"yy/mm/dd"	"89/12/25"
"yy/mm/dd"	"1989, December 25th"
"yy/mm/dd"	"In the year 1989, the month of December, its 25th day"
"dd-mm-yy"	"This 25th day of December, 1989"

---

# RFMTDATE

## Overview

Use `rfmtdate` to convert a date in internal format to a string formatted according to a pattern.

## Syntax

---

```
rfmtdate(jdate, fmtstring, result)
    long jdate;
    char *fmtstring;
    char *result;
```

---

## Explanation

*jdate* is the internal representation of a date as a long integer.

*fmtstring* is a pointer to the character array containing the format pattern for the date returned in *result*.

*result* is a pointer to the character array that receives the formatted date.

## Notes

1. The *fmtstring* date string consists of combinations of the characters *m*, *d*, and *y* as shown in the following example:

---

dd	Day of the month as a 2-digit number (01-31)
ddd	Day of the week as a 3-letter abbreviation (Sun through Sat)
mm	Month as a 2-digit number (01-12)
mmm	Month as a 3-letter abbreviation (Jan through Dec)
yy	Year as a 2-digit number in the 1900s (00-99)
yyyy	Year as a 4-digit number (0001-9999)

---

2. Any other characters in *fmtstring* are reproduced literally in *result*.

### Return Codes

- |       |   |
|-------|---|
| 0     | The conversion was successful.                                  |
| -1210 | The internal date cannot be converted to month-day-year format. |
| -1211 | Program has run out of memory—memory allocation error.          |

### Examples

The examples that follow convert the integer *jdate* that corresponds to December 25, 1989, to a string *result* using the format in *fmtstring*:



---

<b>fmtstring</b>	<b>result</b>
"mmdyy"	122589
"ddmmyy"	251289
"ymmdd"	891225
"yy/mm/dd"	89/12/25
"yy mm dd"	89 12 25
"yy-mm-dd"	89-12-25
"mmm. dd, yyyy"	Dec. 25, 1989
"mmm dd yyyy"	Dec 25 1989
"yyyy dd mm"	1989 25 12
"mmm dd yyyy"	Dec 25 1989
"ddd, mmm. dd, yyyy"	Mon, Dec. 25, 1989
"(ddd) mmm. dd, yyyy"	(Mon) Dec. 25, 1989

---

# RJULMDY

## Overview

The **rjulmdy** routine creates an array of three short integers containing the month, day, and year components corresponding to an internal date.

## Syntax

---

```
rjulmdy(jdate, mdy)
    long jdate;
    short mdy[3];
```

---

## Explanation

*jdate* is the internal representation of the date as a long integer.

*mdy* is an array of short integers, where *mdy*[0] is the month (1 to 12), *mdy*[1] is the day (1 to 31), and *mdy*[2] is the year (1 to 9999).

## Return Codes

= 0 The operation was successful.  
< 0 The operation failed.

# RLEAPYEAR

## Overview

The `rleapyear` routine returns `TRUE` when the argument passed to it is a leap year and `FALSE` when it is not.

## Syntax

---

```
rleapyear(year)  
    int year;
```

---

## Explanation

*year* is an integer.

## Notes

1. The argument *year* must be the year component of a date and not the date itself.
2. The *year* must be expressed in full (1989) and not abbreviated (89).

## Return Codes

`TRUE(1)` The year is a leap year.  
`FALSE(0)` The year is not a leap year.

# RMDYJUL

## Overview

Use `rmdyjul` to create an internal date from three short integers that contain the numeric values for the month, day, and year.

## Syntax

---

```
rmdyjul(mdy, jdate)
    short mdy[3];
    long *jdate;
```

---

## Explanation

*mdy* is an array of short integers, where *mdy*[0] is the month (1 to 12), *mdy*[1] is the day (1 to 31), and *mdy*[2] is the year (1 to 9999).

*jdate* is a pointer to the internal representation of the returned date as a long integer.

## Note

The year must be expressed in full (1989) and not abbreviated (89).

## Return Codes

- |       |   |
|-------|---|
| 0     | The operation was successful.                           |
| -1204 | There was an invalid year component in <i>mdy</i> [2].  |
| -1205 | There was an invalid month component in <i>mdy</i> [0]. |
| -1206 | There was an invalid day component in <i>mdy</i> [1].   |



# RSTRDATE

## Overview

Use **rstrdate** to convert a character string date to a date in internal format.

## Syntax

---

```
rstrdate(str, jdate)
    char *str;
    long *jdate;
```

---

## Explanation

*str* is a pointer to the string to be converted.

*jdate* is a pointer to a long integer that receives the converted date.

## Notes

1. The *str* should contain a numeric month, day, and year in that order. Any non-numeric character can be used as a separator between the month, day, and year.
2. The year must be expressed in full (1989) and not abbreviated (89).

## Return Codes

- = 0 The conversion was successful.
- < 0 The conversion failed.

# RTODAY

## Overview

Use **rtoday** to put the system date into internal format.

## Syntax

---

```
rtoday(today)  
    long *today;
```

---

## Explanation

*today* is a pointer to a long integer that receives the current system date in internal format.

## MONEY Type

A column with data type  $\text{MONEY}(m,n)$  is represented as a  $\text{DECIMAL}$  type with a fixed precision  $m$  and a fixed number  $n$  of decimal places. A column declared as  $\text{MONEY}(m)$  is treated as  $\text{DECIMAL}(m, 2)$ . A column declared as  $\text{MONEY}$  without precision or scale is treated as  $\text{DECIMAL}(16, 2)$ . Regardless of the number of parameters,  $\text{MONEY}$  type variables always have fixed-point arithmetic. See the following section, "DECIMAL Type," for further information.

## DECIMAL Type

The data type  $\text{DECIMAL}$  is a machine-independent method for the representation of numbers of up to 32 significant digits, with or without a decimal point, and with exponents in the range  $-128$  to  $+126$ .

$\text{INFORMIX-ESQL/C}$  provides routines that facilitate the conversion of  $\text{DECIMAL}$  type numbers to and from every data type allowed in the C Language.

When you define a column as having the data type  $\text{DECIMAL}(m,n)$ , it has a total of  $m$  ( $\leq 32$ ) significant digits (the precision) and  $n$  ( $\leq m$ ) digits to the right of the decimal point (the scale). When you give values for both  $m$  and  $n$ , the decimal variable has fixed-point arithmetic. All numbers less than  $0.5 \times 10^{-n}$  in absolute value have the value zero. The largest absolute value of a variable of this type that can be stored without an error is  $10^{m-n} - 10^{-n}$ .

The second parameter is optional and, when missing, the variable is treated as a floating decimal. This means that  $\text{DECIMAL}(m)$  variables have a precision of  $m$  and a range in absolute value from  $10^{-128}$  to  $10^{126}$ . When no parameters are designated,  $\text{DECIMAL}$  is treated as  $\text{DECIMAL}(16)$ , a floating decimal.

$\text{DECIMAL}$  type numbers consist of an exponent and a mantissa (or fractional part) in base 100. In normalized form, the first digit of the mantissa must be greater than zero.

When used within a program, DECIMAL type numbers are stored in a C structure of the following type.

---

```
#define DECSIZE 16

struct decimal
{
    short dec_exp;
    short dec_pos;
    short dec_ndgts;
    char  dec_dgts[DECSIZE];
};

typedef struct decimal dec_t;
```

---

The **decimal** structure and the typedef **dec\_t** are found in the header file **decimal.h**. Include this file in all C source files that use any of the decimal routines:

```
#include <decimal.h>
```

The **decimal** structure has four parts:

- |                  |   |
|------------------|---|
| <b>dec_exp</b>   | holds the exponent of the normalized DECIMAL type number. This exponent represents a power of 100.  |
| <b>dec_pos</b>   | holds the sign of the DECIMAL type number (1 when the number is zero or greater; 0 when less than zero).  |
| <b>dec_ndgts</b> | contains the number of base 100 significant digits of the DECIMAL type number.  |
| <b>dec_dgts</b>  | is a character array that holds the significant digits of the normalized DECIMAL type number ( <b>dec_dgts[0]</b> != 0). Each character in the array is a one-byte binary number in base 100. <b>dec_ndgts</b> contains the number of significant digits in <b>dec_dgts</b> . |

All operations on DECIMAL type numbers take place through the routines provided in **INFORMIX-ESQL/C** and described in the following section. Any other operations, modifications, or analyses can produce unpredictable results.



## ***DECIMAL Type Routines***

The following C function calls are available in INFORMIX-ESQL/C to treat DECIMAL type numbers:

---

<b>deccvasc</b>	Convert C char type to DECIMAL type
<b>dectoasc</b>	Convert DECIMAL type to C char type
<b>deccvint</b>	Convert C int type to DECIMAL type
<b>dectoint</b>	Convert DECIMAL type to C int type
<b>deccvlong</b>	Convert C long type to DECIMAL type
<b>dectolong</b>	Convert DECIMAL type to C long type
<b>deccvdbl</b>	Convert C double type to DECIMAL type
<b>dectodbl</b>	Convert DECIMAL type to C double type
<b>decadd</b>	Add two decimal numbers
<b>decsub</b>	Subtract two decimal numbers
<b>decmul</b>	Multiply two decimal numbers
<b>decdiv</b>	Divide two decimal numbers
<b>deccmp</b>	Compare two decimal numbers
<b>deccopy</b>	Copy a decimal number
<b>dececv</b>	Convert decimal value to ASCII string
<b>decfcvt</b>	Convert decimal value to ASCII string
<b>decround</b>	Round a decimal number
<b>dectrunc</b>	Truncate a decimal number

---

# DECCVASC

## Overview

The **deccvasc** routine converts a value held as printable characters in a C char type into a DECIMAL type number.

## Syntax

---

```
deccvasc(cp, len, np)
    char *cp;
    int len;
    dec_t *np;
```

---

## Explanation

<i>cp</i>	is a pointer to a string that holds the value to be converted.
<i>len</i>	is the length of the string.
<i>np</i>	is a pointer to the decimal structure where the result of the conversion is placed.

## Notes

1. Leading spaces in the character string are ignored.
2. The character string can have a leading sign, either + or -, a decimal point, and digits to the right of the decimal point.
3. The character string may contain an exponent preceded by either *e* or *E*. The exponent can be preceded by a sign, either + or -.

## Return Codes

0	The conversion was successful.
-1200	The number is too large to fit into a DECIMAL type (overflow).
-1201	The number is too small to fit into a DECIMAL type (underflow).
-1213	The string has non-numeric characters.
-1216	The string has bad exponent.

## Example

---

```
#include <decimal.h>

char input[80];
dec_t number;
.
.
.
/* get input from terminal */
getline(input);

/* convert input into decimal number */
deccvasc(input, 32, &number);
```

---

# DECTOASC

## Overview

The **dectoasc** routine converts a DECIMAL type number to an ASCII string.

## Syntax

---

```
dectoasc(np, cp, len, right)
    dec_t *np;
    char *cp;
    int len;
    int right;
```

---

## Explanation

- |              |   |
|--------------|---|
| <i>np</i>    | is a pointer to the decimal structure whose associated decimal value is converted into an ASCII string. |
| <i>cp</i>    | is a pointer to the beginning of the character buffer to hold the ASCII string.                         |
| <i>len</i>   | is the maximum length in bytes of the string buffer.  |
| <i>right</i> | is an integer indicating the number of decimal places to the right of the decimal point.                |

## Notes

1. If *right* = -1, the number of decimal places is determined by the decimal value of *\*np*.
2. If the number does not fit into a character string of length *len*, **dectoasc** converts the number to an exponential notation. If the number still does not fit, the string is filled with asterisks. If the number is shorter than the string, it is left justified and padded on the right with blanks.



3. Because the ASCII string returned by `dectoasc` is not null-terminated, your program must add a null character to the string before printing it.

## Return Codes

- 0      The conversion was successful.
- 1     The conversion failed.

## Example

---

```
#include <decimal.h>

char input[80];
char output[16];
dec_t number;
.
.
.

/* get input from terminal */
getline(input);

/* convert input into decimal number */
deccvasc(input, 32, &number);

/* convert number to ASCII string*/
dectoasc(&number, output, 15, 1);

/* add null character to end of string prior to printing*/
output[15] = '\0';

/* print the value just entered */
printf("You just entered %s", output);
```

---

# DECCVINT

## Overview

Use `deccvint` to convert a C type `INT` into a `DECIMAL` type number.

## Syntax

---

```
deccvint(integer, np)
    int integer;
    dec_t *np;
```

---

## Explanation

*integer* is the integer to be converted.

*np* is a pointer to a decimal structure where the result is placed.

## Example

---

```
#include <decimal.h>

dec_t decnum;

/* convert the integer value -999
 * into a DECIMAL type number
 */
deccvint(-999, &decnum);
```

---

# DECTOINT

## Overview

Use **dectoint** to convert a DECIMAL type number into a C type INT.

## Syntax

---

```
dectoint(np, ip)
    dec_t *np;
    int *ip;
```

---

## Explanation

*np* is a pointer to a decimal structure whose value is converted to an integer.

*ip* is a pointer to the integer.

## Return Codes

0	The conversion was successful.
-1200	The magnitude of the DECIMAL type number > 32767.

## Example

---

```
#include <decimal.h>

dec_t mydecimal;
int myinteger;

/* convert the value in
 * mydecimal into an integer
 * and place the results in
 * the variable myinteger.
 */
dectoint(&mydecimal, &myinteger);
```

---

# DECCVLONG

## Overview

Use **deccvlong** to convert a C type long value into a DECIMAL type number.

## Syntax

---

```
deccvlong(lng, np)
    long lng;
    dec_t *np;
```

---

## Explanation

*lng* is the long value that is converted into a DECIMAL type value.

*np* is a pointer to a decimal structure that holds the DECIMAL type number.

## Examples

---

```
#include <decimal.h>

dec_t mydecimal;
long mylong;

/* Set the decimal structure
 * mydecimal to 37.
 */
deccvlong(37L, &mydecimal);

mylong = 123456L;
/* Convert the variable mylong into
 * a DECIMAL type number held in
 * mydecimal.
 */
deccvlong(mylong, &mydecimal);
```

---



# DECTOLONG

## Overview

Use **dectolong** to convert a DECIMAL type number into a C type long.

## Syntax

---

```
dectolong(np, lngp)
    dec_t *np;
    long *lngp;
```

---

## Explanation

*np* is a pointer to a decimal structure.

*lngp* is a pointer to a long integer where the result of the conversion is placed.

## Return Codes

0 The conversion was successful.

-1200 The magnitude of the DECIMAL type number > 2,147,483,647.

## Example

---

```
#include <decimal.h>

dec_t mydecimal;
long mylong;

/* convert the DECIMAL type value
 * held in the decimal structure
 * mydecimal to a long pointed to
 * by mylong.
 */
dectolong(&mydecimal, &mylong);
```

---

# DECCVDBL

## Overview

Use **deccvdbl** to convert a C type double into a DECIMAL type number.

## Syntax

---

```
deccvdbl(dbl, np)
    double dbl;
    dec_t *np;
```

---

## Explanation

*dbl* is the double value that is converted into a DECIMAL type value.

*np* is a decimal structure that contains the DECIMAL type number.

## Examples

---

```
#include <decimal.h>

dec_t mydecimal;
double mydouble;

/* Set the decimal structure
 * mydecimal to 3.14159.
 */
deccvdbl(3.14159, &mydecimal);

mydouble = 123456.78;

/* Convert the variable mydouble into
 * a DECIMAL type number held in
 * mydecimal.
 */
deccvdbl(mydouble, &mydecimal);
```

---

# DECTODBL

## Overview

Use **dectodbl** to convert a DECIMAL type number into a double.

## Syntax

---

```
dectodbl(np, dblp)
    dec_t *np;
    double *dblp;
```

---

## Explanation

*np* is a pointer to a decimal structure.

*dblp* is a pointer to a double where the result of the conversion is placed.

## Note

Depending upon the floating-point format of the host machine, the conversion of a DECIMAL type number to a double can result in the loss of precision.

## Example

---

```
#include <decimal.h>

dec_t mydecimal;
double mydouble;

/* convert the DECIMAL type value
 * held in the decimal structure
 * mydecimal to a double pointed to
 * by mydouble.
 */
dectodbl(&mydecimal, &mydouble);
```

---

# DECADD, DECSUB, DECMUL, and DECDIV

## Overview

The decimal arithmetic routines take pointers to three decimal structures as parameters. The first two decimal structures hold the operands of the arithmetic function. The third decimal structure is where the result is placed.

## Syntax

---

<code>decadd(n1, n2, result)</code>	<code>/* result = n1 + n2 */</code>
<code>  dec_t *n1;</code>	
<code>  dec_t *n2;</code>	
<code>  dec_t *result;</code>	
<code>decsub(n1, n2, result)</code>	<code>/* result = n1 - n2 */</code>
<code>  dec_t *n1;</code>	
<code>  dec_t *n2;</code>	
<code>  dec_t *result;</code>	
<code>decmul(n1, n2, result)</code>	<code>/* result = n1 * n2 */</code>
<code>  dec_t *n1;</code>	
<code>  dec_t *n2;</code>	
<code>  dec_t *result;</code>	
<code>decdiv(n1, n2, result)</code>	<code>/* result = n1 / n2 */</code>
<code>  dec_t *n1;</code>	
<code>  dec_t *n2;</code>	
<code>  dec_t *result;</code>	

---

## Explanation

<i>n1</i>	is a pointer to the decimal structure of the first operand.
<i>n2</i>	is a pointer to the decimal structure of the second operand.
<i>result</i>	is a pointer to the decimal structure of the result of the operation.



## Note

The *result* can be the same as either *n1* or *n2*.

## Return Codes

0	The operation was successful.
-1200	The operation resulted in overflow.
-1201	The operation resulted in underflow.
-1202	The operation attempted to divide by zero.

# DECCMP

## Overview

Use **deccmp** to compare two DECIMAL type numbers.

## Syntax

---

```
int deccmp(n1, n2)
           dec_t *n1;
           dec_t *n2;
```

---

## Explanation

*n1* is a pointer to the decimal structure of the first number.

*n2* is a pointer to the decimal structure of the second number.

## Return Codes

- 1 The first value is less than the second.
- 0 The two values are the same.
- 1 The first value is greater than the second.

**DECUNKNOWN** Either value is NULL.

# DECCOPY

## Overview

Use `deccopy` to copy one decimal structure to another.

## Syntax

---

```
deccopy(n1, n2)
    dec_t *n1;
    dec_t *n2;
```

---

## Explanation

*n1* is a pointer to the value held in the source decimal structure.

*n2* is a pointer to the destination decimal structure.

# DECECVT and DECFCVT

## Overview

These decimal routines are analogous to the subroutines under ECVT(3) in section three of the UNIX Programmer's Manual. **dececv**t works in the same fashion as **ecvt**(3), and **decfcvt** works in the same fashion as **fcvt**(3). They both convert a decimal value to an ASCII string.

## Syntax

---

```
char *dececv(np, ndigit, decpt, sign)
    dec_t *np;
    int ndigit;
    int *decpt;
    int *sign;
```

```
char *decfcvt(np, ndigit, decpt, sign)
    dec_t *np;
    int ndigit;
    int *decpt;
    int *sign;
```

---

## Explanation

- np* is a pointer to a decimal structure containing the decimal value to be converted.
- ndigit* is, for **dececv**t, the length of the ASCII string. For **decfcvt**, it is the number of digits to the right of the decimal point.
- decpt* is a pointer to an integer that is the position of the decimal point relative to the beginning of the string. A negative value for *\*decpt* means to the left of the returned digits.
- sign* is a pointer to the sign of the result. If the sign of the result is negative, *\*sign* is nonzero; otherwise, it is zero.



## Notes

1. **dececv**t converts the decimal value pointed to by *np* into a null-terminated string of *ndigit* ASCII digits and returns a pointer to the string.
2. The low-order digit is rounded.
3. **decfcvt** is identical to **dececv**t, except that *ndigit* specifies the number of digits to the right of the decimal point instead of the total number of digits.

## Examples

Let *np* point to 12345.67 and suppress all arguments except *ndigit*:

---

<code>dececv</code> t(4)	= "1235"	<code>*decpt</code> = 5
<code>dececv</code> t(10)	= "1234567000"	<code>*decpt</code> = 5
<code>decfcvt</code> (1)	= "123457"	<code>*decpt</code> = 5
<code>decfcvt</code> (3)	= "12345670"	<code>*decpt</code> = 5

---

Now let *np* point to .001234:

---

<code>dececv</code> t(4)	= "1234"	<code>*decpt</code> = -2
<code>dececv</code> t(10)	= "1234000000"	<code>*decpt</code> = -2
<code>decfcvt</code> (1)	= ""	<code>*decpt</code> = -2
<code>decfcvt</code> (3)	= "1"	<code>*decpt</code> = -2

---

# DECROUND

## Overview

Use `decround` to round a DECIMAL type number to fractional digits.

## Syntax

---

```
decround(d, s)
    dec_t *d;
    int *s;
```

---

## Explanation

*d* is a `dec_t` structure for a decimal number whose value is rounded.

*s* is the number of fractional digits the number is rounded to.

## Note

The rounding factor is  $5 \times 10^{-s-1}$ . Rounding is performed by adding the factor to a positive number or by subtracting it from a negative number, and then truncating to *s* digits.

## Examples

---

unrounded	s	rounded	truncated
1.4	0	1.0	1.0
1.5	0	2.0	1.0
1.684	2	1.68	1.68
1.685	2	1.69	1.68
1.685	1	1.7	1.6
1.685	0	2.0	1.0

---

# DECTRUNC

## Overview

Use **dectrunc** to truncate to fractional digits a DECIMAL type number that has been rounded.

## Syntax

```
dectrunc(d, s)
    dec_t *d;
    int *s;
```

## Explanation

- d* is a dec\_t structure for a rounded decimal number whose value is truncated.
- s* is the number of fractional digits the number is truncated to.

## Examples

unrounded	s	rounded	truncated
1.4	0	1.0	1.0
1.5	0	2.0	1.0
1.684	2	1.68	1.68
1.685	2	1.69	1.68
1.685	1	1.7	1.6
1.685	0	2.0	1.0

# Numeric Formatting Routines

You can use special run-time functions to format a numeric expression according to a specific pattern. These formatting routines let you line up decimal points, right- or left-justify numbers, put negative numbers in parentheses, and perform other formatting functions.

The following functions are included in the libraries for formatting numeric expressions in **INFORMIX-ESQL/C**:

---

<b>rfmtdec</b>	Convert decimal to string
<b>rfmtdouble</b>	Convert double to string
<b>rfmtlong</b>	Convert long integer to string

---

These formatting routines and their syntax are shown on the following pages. General formatting rules and examples of formatted results follow the routines.



# RFMTDEC

## Overview

The **rfmtdec** function converts a **dec\_t** in internal format to a character string formatted according to a pattern.

## Syntax

---

```
rfmtdec(dec, format, outbuf)
    dec_t *dec;
    char *format;
    char *outbuf;
```

---

## Explanation

<i>dec</i>	is the number to be formatted.
<i>format</i>	is the address of the formatted string.
<i>outbuf</i>	is the address of the buffer to receive the formatted string.

## Return Codes

0	The conversion was successful.
-1211	Program has run out of memory—memory allocation error.
-1217	The format string is too large.

# RFMTDOUBLE

## Overview

The **rfmtdouble** function converts a double in internal format to a character string formatted according to a pattern.

## Syntax

---

```
rfmtdouble(dvalue, format, outbuf)  
    double dvalue;  
    char *format;  
    char *outbuf;
```

---

## Explanation

<i>dvalue</i>	is the number to be formatted.
<i>format</i>	is the address of the formatted string.
<i>outbuf</i>	is the address of the buffer to receive the formatted string.

## Return Codes

0	The conversion was successful.
-1211	Program has run out of memory—memory allocation error.
-1217	The format string is too large.

# RFMTLONG

## Overview

The **rfmtlong** function converts a long in internal format to a character string formatted according to a pattern.

## Syntax

---

```
rfmtlong(lvalue, format, outbuf)  
    long lvalue;  
    char *format;  
    char *outbuf;
```

---

## Explanation

<i>lvalue</i>	is the number to be formatted.
<i>format</i>	is the address of the formatted string.
<i>outbuf</i>	is the address of the buffer to receive the formatted string.

## Return Codes

0	The conversion was successful.
-1211	Program has run out of memory—memory allocation error.
-1217	The format string is too large.

## Formatting Numeric Strings

The numeric expression format string consists of combinations of the following characters: \* & # < , . - + ( ) \$. Descriptions of these characters follow.

The characters - + ( ) \$ will *float*. When a character floats, multiple leading occurrences of the character appear as a single character as far to the right as possible, without interfering with the number that is being displayed.

- \* This character fills with asterisks any positions in the display field that would otherwise be blank.
- & This character fills with zeros any positions in the display field that would otherwise be blank.
- # This character does not change any blank positions in the display field. Use this character to specify a maximum width for a field.
- < This character causes the numbers in the display field to be left-justified.
- , This character is a literal. It displays as a comma, but only if there is a number to its left.
- . This character is a literal that displays as a period. You can have only one period in a format string.
- This character is a literal. It displays as a minus sign when *expr1* is less than zero. When you group several in a row, a single minus sign floats to the rightmost position without interfering with the number being printed.
- + This character is a literal. It displays as a plus sign when *expr1* is greater than or equal to zero and as a minus sign when it is less than zero. When you group several in a row, a single plus sign floats to the rightmost position without interfering with the number being printed.



- ( This character is a literal. It displays as a left parenthesis before a negative number. It is the accounting parenthesis that is used in place of a minus sign to indicate a negative number. When you group several in a row, a single left parenthesis floats to the rightmost position without interfering with the number being printed.
- ) This is the accounting parenthesis that is used in place of a minus sign to indicate a negative number. A single one of these characters generally closes a format string that begins with a left parenthesis.
- \$ This character is a literal. It displays as a dollar sign. When you group several in a row, a single dollar sign floats to the rightmost position without interfering with the number being printed.

The next four pages show example format strings for numeric expressions.

## Example Format Strings

Format String	Numeric Value	Formatted Result
"####"	0	bbbb
"&&&&"	0	0000
"\$\$\$\$"	0	bbbb\$
"*****"	0	*****
"<<<<"	0	(null string)
"##,###"	12345	12,345
"##,###"	1234	b1,234
"##,###"	123	bbb123
"##,###"	12	bbbb12
"##,###"	1	bbbbb1
"##,###"	-1	bbbbb1
"##,###"	0	bbbbbb
"&&, &&&"	12345	12,345
"&&, &&&"	1234	01,234
"&&, &&&"	123	000123
"&&, &&&"	12	000012
"&&, &&&"	1	000001
"&&, &&&"	-1	000001
"&&, &&&"	0	000000
"\$\$, \$\$\$"	12345	***** (overflow)
"\$\$, \$\$\$"	1234	\$1,234
"\$\$, \$\$\$"	123	bb\$123
"\$\$, \$\$\$"	12	bbb\$12
"\$\$, \$\$\$"	1	bbbb\$1
"\$\$, \$\$\$"	-1	bbbb\$1
"\$\$, \$\$\$"	0	bbbbbb\$
"*, ***"	12345	12,345
"*, ***"	1234	*1,234
"*, ***"	123	***123
"*, ***"	12	****12
"*, ***"	1	*****1
"*, ***"	0	*****

This table uses the character b to represent a blank or space.

## Example Format Strings

Format String	Numeric Value	Formatted Result
"##,###.##"	12345.67	12,345.67
"##,###.##"	1234.56	b1,234.56
"##,###.##"	123.45	bbb123.45
"##,###.##"	12.34	bbbb12.34
"##,###.##"	1.23	bbbbb1.23
"##,###.##"	0.12	bbbbbb.12
"##,###.##"	0.01	bbbbbb.01
"##,###.##"	-0.01	bbbbbb.01
"##,###.##"	-1	bbbbb1.00
"&&, &&&. &&"	12345.67	12,345.67
"&&, &&&. &&"	1234.56	01,234.56
"&&, &&&. &&"	123.45	000123.45
"&&, &&&. &&"	0.01	000000.01
"\$\$,\$\$\$.\$\$"	12345.67	***** (overflow)
"\$\$,\$\$\$.\$\$"	1234.56	\$1,234.56
"\$\$,\$\$\$.##"	0.00	\$ .00
"\$\$,\$\$\$.##"	1234.00	\$1,234.00
"\$\$,\$\$\$.&&"	0.00	\$ .00
"\$\$,\$\$\$.&&"	1234.00	\$1,234.00
"-##,###.##"	-12345.67	-12,345.67
"-##,###.##"	-123.45	-bbb123.45
"-##,###.##"	-12.34	-bbbb12.34
"--#,###.##"	-12.34	-bbb12.34
"---,###.##"	-12.34	-bb12.34
"---,-##.##"	-12.34	-12.34
"---,-#.##"	-1.00	-1.00
"-##,###.##"	12345.67	12,345.67
"-##,###.##"	1234.56	1,234.56
"-##,###.##"	123.45	123.45
"-##,###.##"	12.34	12.34
"--#,###.##"	12.34	12.34
"---,###.##"	12.34	12.34
"---,-##.##"	12.34	12.34
"---,---.##"	1.00	1.00
"---,---.-"	-.01	-.01
"---,---.&&"	-.01	-.01

This table uses the character b to represent a blank or space.

## Example Format Strings

Format String	Numeric Value	Formatted Result
"-\$\$\$,\$\$\$.&&"	-12345.67	-\$12,345.67
"-\$\$\$,\$\$\$.&&"	-1234.56	-b\$1,234.56
"-\$\$\$,\$\$\$.&&"	-123.45	-bbb\$123.45
"--\$\$,\$\$\$.&&"	-12345.67	-\$12,345.67
"--\$\$,\$\$\$.&&"	-1234.56	-\$1,234.56
"--\$\$,\$\$\$.&&"	-123.45	-bb\$123.45
"--\$\$,\$\$\$.&&"	-12.34	-bbb\$12.34
"--\$\$,\$\$\$.&&"	-1.23	-bbbb\$1.23
"----,--\$.&&"	-12345.67	-\$12,345.67
"----,--\$.&&"	-1234.56	-\$1,234.56
"----,--\$.&&"	-123.45	-\$123.45
"----,--\$.&&"	-12.34	-\$12.34
"----,--\$.&&"	-1.23	-\$1.23
"----,--\$.&&"	-.12	-\$12
"\$***,***.&&"	12345.67	\$*12,345.67
"\$***,***.&&"	1234.56	\$**1,234.56
"\$***,***.&&"	123.45	\$***123.45
"\$***,***.&&"	12.34	\$*****12.34
"\$***,***.&&"	1.23	\$*****1.23
"\$***,***.&&"	.12	\$*****.12
"(\$\$\$,\$\$\$.&&)"	-12345.67	(\$12,345.67)
"(\$\$\$,\$\$\$.&&)"	-1234.56	(b\$1,234.56)
"(\$\$\$,\$\$\$.&&)"	-123.45	(bbb\$123.45)
"((\$\$,,\$\$\$.&&)"	-12345.67	(\$12,345.67)
"((\$\$,,\$\$\$.&&)"	-1234.56	(\$1,234.56)
"((\$\$,,\$\$\$.&&)"	-123.45	(bb\$123.45)
"((\$\$,,\$\$\$.&&)"	-12.34	(bbb\$12.34)
"((\$\$,,\$\$\$.&&)"	-1.23	(bbbb\$1.23)
"(((,((\$.&&)"	-12345.67	(\$12,345.67)
"(((,((\$.&&)"	-1234.56	(\$1,234.56)
"(((,((\$.&&)"	-123.45	(\$123.45)
"(((,((\$.&&)"	-12.34	(\$12.34)
"(((,((\$.&&)"	-1.23	(\$1.23)
"(((,((\$.&&)"	-.12	(\$12)

This table uses the character b to represent a blank or space.



## Example Format Strings

Format String	Numeric Value	Formatted Result
"(\$\$\$,\$\$\$.&&)"	12345.67	\$12,345.67
"(\$\$\$,\$\$\$.&&)"	1234.56	\$1,234.56
"(\$\$\$,\$\$\$.&&)"	123.45	\$123.45
"((\$\$,\$\$\$.&&)"	12345.67	\$12,345.67
"((\$\$,\$\$\$.&&)"	1234.56	\$1,234.56
"((\$\$,\$\$\$.&&)"	123.45	\$123.45
"((\$\$,\$\$\$.&&)"	12.34	\$12.34
"((\$\$,\$\$\$.&&)"	1.23	\$1.23
"((((,(\$.&&)"	12345.67	\$12,345.67
"((((,(\$.&&)"	1234.56	\$1,234.56
"((((,(\$.&&)"	123.45	\$123.45
"((((,(\$.&&)"	12.34	\$12.34
"((((,(\$.&&)"	1.23	\$1.23
"((((,(\$.&&)"	.12	\$.12
"<<<,<<<"	12345	12,345
"<<<,<<<"	1234	1,234
"<<<,<<<"	123	123
"<<<,<<<"	12	12

# DATETIME and INTERVAL Type Routines

The following C function calls are available to treat DATETIME and INTERVAL host variables:

---

<b>dttoasc</b>	Convert a DATETIME to character string
<b>intoasc</b>	Convert an INTERVAL to character string
<b>dtevasc</b>	Convert a character string to DATETIME
<b>incvasc</b>	Convert a character string to INTERVAL
<b>dtcurrent</b>	Get current date and time
<b>dtextend</b>	Change qualifier of DATETIME

---

These routines are shown alphabetically on the following pages. Detailed information on DATETIME and INTERVAL manipulation follows the routines.

In addition to these functions, an *include* file named **datetime.h** is included with the INFORMIX-ESQL/C libraries. This file, which defines the data structures, also defines the following names and macro functions (which are required only when working directly with qualifiers in binary form):

---

<b>TU_YEAR</b>	name for qualifier field
<b>TU_MONTH</b>	name for qualifier field
<b>TU_DAY</b>	name for qualifier field
<b>TU_HOUR</b>	name for qualifier field
<b>TU_MINUTE</b>	name for qualifier field
<b>TU_SECOND</b>	name for qualifier field
<b>TU_FRAC</b>	name for leading qualifier field of FRACTION
<b>TU_Fn</b>	names for datetime ending fields of "FRACTION(n)", for n from 1 to 5
<b>TU_START(q)</b>	returns leading field number from qualifier q
<b>TU_END(q)</b>	returns trailing field number from qualifier q
<b>TU_DTENCODE(f,t)</b>	composes a DATETIME qualifier from first field number f and trailing field number t
<b>TU_IENCODE(p,f,t)</b>	composes an INTERVAL qualifier from first field number f with precision p and trailing field number t

---

# DTCURRENT

## Overview

Use **dtcurrent** to assign the current date and/or time to a DATETIME variable.

## Syntax

---

```
dtcurrent(d)  
    dtime_t *d;
```

---

## Explanation

*d* is the address of an initialized **dtime\_t** host variable.

## Notes

1. If the variable's qualifier is set to zero (or any invalid qualifier), it will be initialized to *year to fraction(3)*.
2. If the variable contains a valid qualifier, the current date and time will be extended to agree with the qualifier.

## Examples

These statements set the variable *timewarp* to the current date.

---

```
$datetime year to day timewarp;  
dtcurrent(&timewarp);
```

---

These statements set the variable *now* to the current time to the nearest millisecond:

---

```
now.dt_qual = TU_DTENCODE(TU_HOUR, TU_F3);  
dtcurrent(&now);
```

---



# DTCVASC

## Overview

Use `dctvasc` to convert a string to a DATETIME value.

## Syntax

---

```
dctvasc(str,d)  
    char *str;  
    dttime_t *d;
```

---

## Explanation

*str* is the address of a string of digits and field delimiters.

*d* is the address of an initialized `dttime_t` variable.

## Notes

1. The variable must be initialized to the desired qualifier.
2. The input string may have leading and trailing spaces. However, from the first significant digit to the last, the only characters accepted are digits and delimiters appropriate to the fields implied by the qualifier.
3. If a year value is given as one or two digits, 1900 will be added to it.
4. If the input string is acceptable, the value is set in the variable and the function returns zero. Otherwise, it does not change the variable and returns a negative error code.

## Return Codes

- 1260 It is not possible to convert between the specified types.
- 1261 Too many digits in the first field of datetime or interval.
- 1262 Non-numeric character in datetime or interval.
- 1263 A field in a datetime or interval is out of range.
- 1264 Extra characters at the end of a datetime or interval.
- 1265 Overflow occurred on a datetime or interval operation.
- 1266 Intervals or datetimes are incompatible for the operation.
- 1267 The result of a datetime computation is out of range.
- 1268 Invalid datetime qualifier.

## Example

Here the variable *columbus* is initialized to Columbus's birthday, 1989:

---

```
$datetime year to day columbus;  
dtcvasc("89-10-9",&columbus);
```

---

# DTEXTEND

## Overview

Use **dtextend** to copy a DATETIME value under a different qualifier.

## Syntax

---

```
dtextend(id,od)
      dttime_t *id, *od;
```

---

## Explanation

*id* is the address of a variable to be copied.

*od* is the address of a variable with a valid qualifier.

## Notes

1. The field digits of *id* are copied to *od*, with the copy controlled by the qualifier of *od*.
2. Fields in *id* that are not included by *od*'s qualifier are disregarded.
3. Fields in *od* that are not present in *id* are filled in as follows:
  - Fields to the left of the most significant field in *id* are filled in from the current time and date.
  - Fields to the right of the least significant field in *id* are filled in with zeros.

## Return Code

-1268 Invalid datetime qualifier.

## Example

In these statements, a variable named *xmas* is set up with the date of Christmas for the current year. The **dtextend** function is used to generate the current year.

---

```
$datetime work, xmas;  
work.dt_qual = TU_DTENCODE(TU_MONTH, TU_DAY);  
dctvasc("12-25",&work);  
xmas.dt_qual = TU_DTENCODE(TU_YEAR, TU_DAY);  
dtextend(&work,&xmas);
```

---



# DTTOASC

## Overview

Use **dttoasc** to convert the field values of a DATETIME variable to an ASCII string.

## Syntax

---

```
dttoasc(d,str)
      dttime_t *d;
      char *str;
```

---

## Explanation

*d* is the address of an initialized **dttime\_t** variable.

*str* is the address of space for a string.

## Notes

1. The digits of the fields of the variable will be converted to ASCII and copied to the output with delimiters (a hyphen, space, colon, or period) between them.
2. The output does *not* include the qualifier or the parentheses that are used to delimit a DATETIME literal in an SQL statement.
3. The output will include one byte for each delimiter (a hyphen, space, colon, or period) plus the fields with sizes as follows:

year: 4 digits

fraction of DATETIME as specified by precision

all other fields: two digits

The maximum length of output would be produced from a DATETIME qualified as *year to fraction(5)*. It would contain 19 digits, 6 delimiters, and the terminating null, for a total of 26 bytes.

4. If the variable has not been initialized, the function will return an unpredictable value, but one not exceeding 26 bytes.

### Example

---

```
$datetime days to seconds x;  
char buff[20];  
dttoasc(&x, buff);
```

---

# INCVASC

## Overview

Use **incvasc** to convert a string to an INTERVAL value.

## Syntax

---

```
incvasc(str,i)  
    char *str;  
    intrvl_t *i;
```

---

## Explanation

*str* is the address of a string of digits and field delimiters.

*i* is the address of an initialized **intrvl\_t** variable.

## Notes

1. The variable must be initialized to the desired qualifier.
2. The input string may have leading and trailing spaces. However, from the first significant digit to the last, the only characters accepted are digits and delimiters appropriate to the fields implied by the qualifier.
3. If the input string is acceptable, the value is set in the variable and the function returns zero. Otherwise, it does not change the variable and returns a negative error code.

## Return Codes

- 1260 It is not possible to convert between the specified types.
- 1261 Too many digits in the first field of datetime or interval.
- 1262 Non-numeric character in datetime or interval.
- 1263 A field in a datetime or interval is out of range.
- 1264 Extra characters at the end of a datetime or interval.
- 1265 Overflow occurred on a datetime or interval operation.
- 1266 Intervals or datetimes are incompatible for the operation.
- 1267 The result of a datetime computation is out of range.
- 1268 Invalid datetime qualifier.

## Example

---

```
$interval days to seconds x;  
incvasc("20 3:10:35",&x);
```

---



# INTOASC

## Overview

Use **intoasc** to convert the field values of an **INTERVAL** variable to an ASCII string.

## Syntax

---

```
intoasc(i,str)
      intrvl t *i;
      char *str;
```

---

## Explanation

*i* is the address of an initialized **intrvl\_t** variable.

*str* is the address of space for a string.

## Notes

1. The digits of the fields of the variable will be converted to ASCII and copied to the output with delimiters (a hyphen, space, colon, or period) between them.
2. The output does *not* include the qualifier or the parentheses that are used to delimit an interval literal in an SQL statement.
3. The output will include one byte for each delimiter (a hyphen, space, colon, or period) plus the fields with sizes as follows:

leading field as specified by precision  
fraction as specified by precision  
all other fields: two digits

The maximum length of output would be produced from an interval qualified as *day(5) to fraction(5)*. It would contain 16 digits, 4 delimiters, and the terminating null for a total of 21 bytes.

4. If the variable has not been initialized, the function will return an unpredictable value, but one not exceeding 21 bytes.

### Example

---

```
$interval days to seconds x;  
char buff[20];  
intoasc(&x, buff);
```

---

## DATETIME and INTERVAL Types

The DATETIME data type encodes a time to a particular precision. The precision is expressed by a qualifier, and the qualifier is an integral part of the data type. As a host variable, a DATETIME value is represented in a structure of type `dtm_t`:

---

```
typedef struct dtm_t {
    short dt_qual;
    dec_t dt_dec;
} dtm_t;
```

---

The qualifier of the value is represented in the `dt_qual` field, and the digits of the fields of the value are stored in `dt_dec`.

The INTERVAL data type encodes an interval of time to a particular precision. The precision is expressed by a qualifier and, as with DATETIME, the qualifier is an integral part of the data type. As a host variable, an INTERVAL value is represented in a structure of type `intrvl_t`:

---

```
typedef struct intrvl_t {
    short in_qual;
    dec_t in_dec;
} intrvl_t;
```

---

These structures, along with a number of macro definitions for use in composing qualifier values, are contained in the *include* file `datetime.h`. The type `dec_t` that is a component of these structures is defined in the file `decimal.h`. More detailed information on manipulating DATETIME and INTERVAL data types is provided in Appendix H.

A DATETIME or INTERVAL is stored as a decimal number with a scale factor of zero and a precision equal to the number of digits implied by its qualifier. Once you know the precision and scale, you know the storage format. For example, a table column defined as *datetime year to day* contains four digits for year, two digits for month, and two digits for day, for a total of eight digits. It is thus stored as if it were DECIMAL(8,0).

## DATETIME Columns

A DATETIME column holds a value that represents a moment in time. It can indicate any time value from a year through a fraction of a second. You establish the precision of the DATETIME column when you create the column. For example,

---

```
create table mytable (mydtime datetime day to minute)
```

---

creates a table named **mytable** that contains a DATETIME column called **mydtime**. The **mydtime** column stores a point in time with the following precision: the day, hour, and minute.

The DATETIME data type is composed of a contiguous sequence of fields. Here is the syntax for a DATETIME definition:

---

```
column-name DATETIME first TO last
```

---

where *first* and *last* are fields taken from the following list:

Field	Valid Entries
YEAR	A year numbered from 1 to 9999.
MONTH	A month numbered from 1 to 12.
DAY	A day numbered from 1 to 31.
HOURL	An hour numbered from 0 (midnight) to 23.
MINUTE	A minute numbered from 0 to 59.
SECOND	A second numbered from 0 to 59.
FRACTION	A decimal fraction of a second with up to 5 digits of precision. The default precision is 3 digits (thousandths of a second).



A DATETIME column need not include all fields from YEAR to FRACTION; it can be a subset of fields or even a single field. You can define a DATETIME column to include only those fields you need, such as a column that includes just MONTH, DAY, and HOUR.

The fields you include must be contiguous. That is, they must include all fields in sequence that make up a single point in time. For example, you cannot include just MONTH and HOUR in a column definition; you must also include DAY.

Operations involving DATETIME values that do not include precision up to YEAR use the current date to supply any additional precision required. When the first field is DAY and the current month is less than 31 days, you can encounter unexpected results. For example, assume it is February, and you want to insert data left over from January 31 into the table created by the following CREATE TABLE statement:

---

```
create table mytable (mytime datetime day to minute)
insert into mytable values (datetime(31 12:30) day to minute)
```

---

Because the column **mytime** does not include the month or year, the current month and year are used to evaluate whether the inserted value is within acceptable bounds. February has only 28 (or 29) days, so no value for DAY can be larger than 28 (or 29). Thus, this INSERT statement would fail because the value 31 for DAY is out of range for February. If your application requires that DATETIME inserts span months in this way, you should always define the DATETIME column with precision up to at least MONTH.

See Appendix H, “Working with DATETIME and INTERVAL Data,” for more information.

## DATETIME *first TO last*

This is a moment in time with the precision *first to last*. A DATETIME column consists of a contiguous sequence of the following fields: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and FRACTION(*n*) of a second.

The values for the fields are entered as integers and are separated by delimiters. The following delimiters are used with DATETIME values:

Delimiter	Placement in DATETIME Expression
hyphen	Between the YEAR and MONTH, and the MONTH and DAY portions of the value.
space	Between the DAY and HOUR portion of the value.
colon	Between the HOUR and MINUTE, and the MINUTE and SECOND portions of the value.
decimal point	Between the SECOND and FRACTION portion of the value.

All field values are 2-digit integers, except for the YEAR (4 digits) and FRACTION (*n* digits) fields. If a year is entered as 2 digits, the leading digits 19 are added.

## ***INTERVAL Columns***

Use the INTERVAL data type for columns in which you plan to store values that represent a span or period of time. An INTERVAL value can represent a span of years and months, or it can represent a span of days through a fraction of a second. You establish the precision of the INTERVAL column when you create the column. For example,

---

```
create table mytable (myival interval hour to second)
```

---

creates a table named **mytable** that contains an INTERVAL column called **myival**. The **myival** column stores an interval of time with the following precision: hours, minutes, and seconds.

Like the DATETIME data type, the INTERVAL data type is composed of a contiguous sequence of fields. Here is the syntax for an INTERVAL definition:

---

*column-name* INTERVAL *first* TO *last*

---

where *first* and *last* are fields taken from *one* of the following two lists:

Field	Valid Entries
YEAR	A number of years.
MONTH	A number of months.
OR	
DAY	A number of days.
HOURL	A number of hours.
MINUTE	A number of minutes.
SECOND	A number of seconds.
FRACTION	A decimal fraction of a second, with up to 5 digits of precision. The default precision is 3 digits (thousandths of a second).

**Note:** You cannot define an INTERVAL column to include fields from both lists. An INTERVAL column can contain YEAR and/or MONTH information, or DAY through FRACTION information. Like a DATETIME column, you can define an INTERVAL column to include only those fields you need.

See Appendix H, "Working with DATETIME and INTERVAL Data," for more information.

### **INTERVAL *first* TO *last***

This is a span of time with the precision *first* to *last*. An INTERVAL column consists of a contiguous sequence of *one* of the following two lists of fields: either YEAR and MONTH, or DAY, HOUR, MINUTE, SECOND, and FRACTION(*n*) of a second.

Like the DATETIME variable, the values for the fields in an INTERVAL variable are entered as integers and are separated by delimiters. The following delimiters are used with INTERVAL values:



Delimiter	Placement in INTERVAL Expression
hyphen	Between the YEAR and MONTH portion of the value.
space	Between the DAY and HOUR portion of the value.
colon	Between the HOUR, MINUTE, and SECOND portion of the value.
decimal point	Between the SECOND and FRACTION portion of the value.

## ***FETCHing DATETIME and INTERVAL Values***

### **FETCH DATETIME Values**

When a DATETIME value is FETCHed into a host variable of type `dttime_t`, one of two things is done.

When the `dt_qual` field contains a valid qualifier, the database value is extended to match the qualifier. (*Extending* is the operation of adding or dropping fields of a DATETIME value to make it match a given qualifier. Extending is done in SQL statements with the `EXTEND` function and in INFORMIX-ESQL/C with the `dtextend()` function.)

When the `dt_qual` field does *not* contain a valid qualifier, the database value and its qualifier are both fetched, thus initializing the host variable. Zero is an invalid qualifier, so zero may be stored into the `dt_qual` field whenever the database value is to be FETCHed without extending it.

### **Fetching INTERVAL Values**

When an INTERVAL value is FETCHed into a host `intrvl_t` variable, the `in_qual` field of the variable is tested. If it contains zero (or any invalid qualifier) both the database value and the database qualifier are FETCHed, initializing the variable.

When the variable qualifier is valid, it is checked for compatibility with the database qualifier. (Interval qualifiers are compatible provided that, if one contains a field of **month** or **year**, the other contains only fields of **month** and **year**.)



## Data Conversion When FETCHing

You can automatically convert DATETIME and INTERVAL values between database columns and host variables of character type (**char**, **string**, or **fixchar**). The fields of the DATETIME or INTERVAL value in the database are converted to a character string, which is stored in the host variable. If the host variable is too short the string is truncated, **sqlca.sqlwarn.sqlwarn1** is set to W and the indicator variable (if any) is set to the needed length.

Note that DATETIME and INTERVAL values cannot be FETCHed automatically into number host variables.

## *Storing DATETIME and INTERVAL Values*

When a host variable is used to store a DATETIME or INTERVAL value in the database, it must contain a valid qualifier.

When storing a DATETIME value, the qualifier in the host variable may be different from the qualifier of the database column. The host variable's value will be extended to match the database qualifier. When storing an INTERVAL value, the host variable's qualifier must be compatible with that of the database column.

If the host qualifier is invalid or interval qualifiers are incompatible, a negative error code is set in **sqlca.sqlcode** and the update or insert operation fails.

## Data Conversion When Storing

When a host variable of character type is used to update or insert a DATETIME or INTERVAL value, the database engine tries to convert the characters using the qualifier and type of the database column. If the conversion fails, **sqlca.sqlcode** is set to a negative value and the update or insert operation fails.

Automatic conversion from number and DATE host variables is not supported.

## ***Declaration of DATETIME and INTERVAL***

You may declare a host variable of type DATETIME using the data type datetime followed by an optional DATETIME qualifier:

---

```
$datetime year to day holidays[10];  
$datetime hour to second wins, places, shows;  
$datetime column6;
```

---

When the qualifier is omitted, as in the last example, the default qualifier is zero, meaning “uninitialized.”

You may declare a host variable of type INTERVAL using the data type interval followed by an optional interval qualifier.

---

```
$interval day(3) to day accrued leave, leave_taken;  
$interval hour to second race_length;  
$interval scheduled;
```

---

When the qualifier is omitted, as in the last example, the default qualifier is zero, meaning “uninitialized.”

Because of the multi-word nature of these data types, it is not possible to declare an uninitialized DATETIME or INTERVAL host variable named **year**, **month**, **day**, **hour**, **minute**, **second**, or **fraction**. You should avoid the following declarations:

---

```
$datetime year; /* will cause an error */  
$datetime year to day year, today; /* ambiguous */
```

---

## ***Converting Between DATETIME and DATE***

No functions are provided to convert automatically between the DATETIME and DATE types. You can perform these conversions using existing functions and intermediate strings.

To convert a DATETIME to a DATE, use these steps:

1. Use **dtextend()** to adjust the DATETIME qualifier to *year to day*.
2. Apply **dttoasc()**, creating a character string in the form *yyyy-mm-dd*.
3. Use **rdefmtdate()** with a pattern argument of *yyyy-mm-dd* to convert the string to a DATE.

To convert a DATE into a DATETIME, use these steps:

1. Declare a host variable with a qualifier of *year to day* (alternatively, initialize the qualifier with the value returned by **TU\_DTENCODE(TU\_YEAR,TU\_DAY)**).
2. Use **rfmtdate()** with a pattern of *yyyy-mm-dd* to convert the DATE to a character string.
3. Use **dtevasc()** to convert the character string to a value in the prepared DATETIME variable.
4. If necessary, use **dtextend()** to adjust the DATETIME qualifier.

# **Chapter 5**

## **Library Functions**





## Chapter 5 Table of Contents

Chapter Overview .....	5
Function Descriptions .....	5
BYCMPR.....	6
BYCOPY .....	8
BYFILL.....	9
BYLENG.....	10
LDCHAR.....	11
RDOWNSHIFT.....	12
RGETMSG .....	13
RISNULL .....	14
RSETNULL .....	15
RSTOD .....	16
RSTOI .....	17
RSTOL .....	18
RTYPALIGN .....	19
RTYPMSIZE .....	20
RTYPNAME.....	22
RTYPWIDTH.....	23
RUPSHIFT .....	24
SQLBREAK .....	25
SQLEXIT .....	26
SQLSTART.....	27
STCAT.....	28
STCHAR .....	29
STCMPR .....	30
STCOPY .....	31
STLENG .....	32



# Chapter Overview

This chapter describes the Informix library functions included with INFORMIX-ESQL/C. Use these functions in your C programs. When you use a compiler shell script (**esql**, **cace**, or **cperf**), these functions are linked automatically to your program.

## Function Descriptions

The Informix library extension comprises the routines listed below. Those beginning with **by** act on and return fixed-length strings of bytes. Those beginning with **rst** and **st** (except **stchar**) operate on and return null-terminated strings. Those beginning with **sql** are SQL database engine control routines.

---

<b>bncmpr</b>	Compare two groups of contiguous bytes
<b>bycopy</b>	Copy bytes from one area to another
<b>byfill</b>	Fill specified area with a character
<b>byleng</b>	Count number of bytes in string
<b>ldchar</b>	Copy fixed-length string to null-terminated string
<b>rdownshift</b>	Convert all letters to lowercase
<b>rgetmsg</b>	Convert error message number into message string text
<b>risnull</b>	Check if C variable is null
<b>rsetnull</b>	Set a C variable to null
<b>rstod</b>	Convert string to double
<b>rstoi</b>	Convert string to short
<b>rstol</b>	Convert string to long
<b>rtypalign</b>	Align data on proper type boundary
<b>rtypmsize</b>	Give byte size of SQL data types
<b>rtypname</b>	Convert data type to string
<b>rtypwidth</b>	Give minimum conversion byte size
<b>rupshift</b>	Convert all letters to uppercase
<b>sqlbreak</b>	Send engine a request to stop processing
<b>sqlexit</b>	Terminate an engine process
<b>sqlstart</b>	Start an engine process
<b>stcat</b>	Concatenate one string to another
<b>stchar</b>	Copy null-terminated string to fixed-length string
<b>stcmp</b>	Compare two strings
<b>stcopy</b>	Copy string to another string
<b>stleng</b>	Count number of bytes in string

---



# BYCMPR

## Overview

The **bycmpr** function compares two groups of contiguous bytes for a given length, returning the results of the comparison as the value of the function.

## Syntax

---

```
bycmpr(byte1, byte2, length)  
    char *byte1;  
    char *byte2;  
    int length;
```

---

## Explanation

- byte1* is a pointer to the starting location of the first group of contiguous bytes.
- byte2* is a pointer to the starting location of the second group of contiguous bytes.
- length* is the number of bytes over which the comparison occurs.

## Notes

1. **bycmpr** performs a byte-by-byte comparison of the two groups of contiguous bytes until a difference is found and returns an integer whose sign is the same as that of the difference between the two differing bytes.
2. The bytes of the *byte2* group are subtracted from those of the *byte1* group.

## Return Codes

- =0     The two groups are identical.
- <0     *byte1* group < *byte2* group.
- >0     *byte1* group > *byte2* group.

# BYCOPY

## Overview

The **bycopy** function copies a given number of bytes from one location to another.

## Syntax

---

```
bycopy(from, to, length)
char *from;
char *to;
int length;
```

---

## Explanation

*from* is a pointer to the starting byte of the group of bytes to be copied.

*to* is a pointer to the starting byte of the destination group of bytes.

*length* is the number of bytes to be copied.

## Note

Take care not to overwrite areas of memory adjacent to the area to be copied.

# BYFILL

## Overview

The **byfill** function fills a specified area with one character.

## Syntax

---

```
byfill(to, length, ch)  
    char *to;  
    int length;  
    char ch;
```

---

## Explanation

<i>to</i>	is the starting byte of the memory area to be filled.
<i>length</i>	is the number of times the character is repeated within the area.
<i>ch</i>	is the character that fills the area.

## Note

Take care not to overwrite areas of memory adjacent to the area to be copied.



# BYLENG

## Overview

The **byleng** function returns the number of significant characters in a string, not counting trailing blanks.

## Syntax

---

```
byleng(from, count)  
char *from;  
int count;
```

---

## Explanation

*from* is a pointer to a fixed-length string (not null-terminated).

*count* is the number of bytes in the fixed-length string.

# LDCHAR

## Overview

The **ldchar** function copies a fixed-length string into a null-terminated string with any trailing blanks removed.

## Syntax

---

```
ldchar(from, count, to)
char *from;
char *to;
int count;
```

---

## Explanation

*from* is a pointer to the fixed-length source string.

*count* is the number of bytes in the fixed-length source string.

*to* is a pointer to the first byte of a null-terminated destination string.

# RDOWNSHIFT

## Overview

The **rdownshift** function changes all of the characters within a null-terminated string to lowercase.

## Syntax

---

```
rdownshift(s)  
char *s;
```

---

## Explanation

**s** is a pointer to a null-terminated string.

# RGETMSG

## Overview

The **rgetmsg** function converts an Informix error message number into the corresponding message text string.

## Syntax

---

```
rgetmsg(msgnum,msgstr,lenmsgstr)
    short msgnum;
    char *msgstr;
    short lenmsgstr;
```

---

## Explanation

<i>msgnum</i>	is the error message number.
<i>msgstr</i>	is the message string (output buffer).
<i>lenmsgstr</i>	is the size of the message string.

## Note

The message number is typically one returned in **sqlca.sqlcode**. The **rgetmsg** function uses the system file for error message text (**/usr/informix/msg**).

## Return Codes

0	The conversion was successful.
-1232	Unknown message number.



# RISNULL

## Overview

The **risnull** function checks whether a C variable is null.

## Syntax

---

```
risnull(type, ptrvar)  
    int type;  
    char *ptrvar;
```

---

## Explanation

*type* is an integer corresponding to the data type of a C variable (see Chapter 4).

*ptrvar* is a pointer to the C variable.

## Return Codes

- 1 The variable is null.
- 0 The variable is not null.

# RSETNULL

## Overview

The **rsetnull** function sets a C variable to a value that corresponds to a database NULL value.

## Syntax

---

```
rsetnull(type, ptrvar)  
    int type;  
    char *ptrvar;
```

---

## Explanation

<i>type</i>	is an integer corresponding to the data type of a C variable (see Chapter 4).
<i>ptrvar</i>	is a pointer to the C variable.

# RSTOD

## Overview

The `rstod` function converts a null-terminated string into a double.

## Syntax

---

```
rstod(string, double_val)  
    char *string;  
    double *double_val;
```

---

## Explanation

*string* is a pointer to a null-terminated string.

*double\_val* is a pointer to a double where the result of the function is held.

## Return Codes

<code>=0</code>	The conversion was successful.
<code>!=0</code>	The conversion failed.

# RSTOI

## Overview

The `rstoi` function converts a null-terminated string into an integer.

## Syntax

---

```
rstoi(string, ival)
char *string;
int *ival;
```

---

## Explanation

*string* is a pointer to a null-terminated string.

*ival* is a pointer to an integer where the result of the function is held.

## Notes

1. The legal range of values is from -32767 to 32767.
2. If *string* corresponds to a NULL integer, *ival* points to the representation for a SMALLINT NULL. If you want to convert a string that corresponds to a long integer, use `rstol`. Failure to do so can result in corrupt data representation.

## Return Codes

- `=0`      The conversion was successful.  
`!=0`      The conversion failed.



# RSTOL

## Overview

The **rstol** function converts a null-terminated string into a long integer.

## Syntax

---

```
rstol(string, long_int)  
char *string;  
long *long_int;
```

---

## Explanation

*string* is a pointer to a null-terminated string.

*long\_int* is a pointer to a long integer where the result of the function is held.

## Note

The legal range of values is from -2,147,483,647 to 2,147,483,647.

## Return Codes

=0      The conversion was successful.  
!=0     The conversion failed.

# RTYPALIGN

## Overview

The **rtypalign** function returns the position of the next proper boundary for a variable of the specified data type.

## Syntax

---

```
rtypalign(pos, type)
    int pos;
    int type;
```

---

## Explanation

*pos* is the current position in a buffer.

*type* is the integer code for a C or SQL data type.

## Notes

1. **rtypalign** and **rtypmsize** are useful when setting up an **sqlda** to **FETCH** data into a buffer because you can use the functions to provide machine independence.
2. The value of *type* is returned by **DESCRIBE** into **sqlda.sqlvar->sqltype**.

## Return Code

*n* (>0) The offset of the next proper boundary for a variable of that *type*.

# RTYPMSIZE

## Overview

The **rtypmsize** function returns the number of bytes you must allocate in memory for the specified C or SQL type.

## Syntax

---

```
rtypmsize(sqltype, sqllen)  
    int sqltype;  
    int sqllen;
```

---

## Explanation

*sqltype* is the integer code of the C or SQL type.

*sqllen* is the number of bytes in the data file for the specified SQL type.

## Notes

1. **rtypmsize** and **rtypalign** are useful when setting up an **sqlda** to FETCH data into a buffer because you can use the functions to provide machine independence.
2. **rtypmsize** is designed to be used with the **sqlda** structure returned by a DESCRIBE statement. *sqltype* and *sqllen* correspond to the components of the same name in each **sqlda.sqlvar** structure.
3. For CCHARTYPE and CSTRINGTYPE, **rtypmsize** adds one byte to the number of characters for the null terminator. For CFIXCHARTYPE, there is no null terminator.

## Return Codes

- 0            *sqltype* is not a valid SQL type.
- n* (>0)    The number of bytes required for data type is *n*.



# RTYPNAME

## Overview

The **rtypname** function returns a null-terminated string containing the name of the specified SQL type.

## Syntax

---

```
char *rtypname(sqltype)
      int sqltype;
```

---

## Explanation

*sqltype* is an integer code for one of the SQL types.

## Return Codes

The following values are returned:

<b>sqltype</b>	<b>return string</b>
SQLCHAR	"char"
SQLSMINT	"smallint"
SQLINT	"integer"
SQLFLOAT	"float"
SQLSMFLOAT	"smallfloat"
SQLDECIMAL	"decimal"
SQLSERIAL	"serial"
SQLDATE	"date"
SQLMONEY	"money"
SQLDTIME	"datetime"
SQLINTERVAL	"interval"
invalid type	"" (null string)

# RTYPWIDTH

## Overview

The **rtypwidth** function returns the minimum number of characters required to avoid truncation when converting a value with an SQL type to a character data type.

## Syntax

---

```
rtypwidth(sqltype, sqllen)
    int sqltype;
    int sqllen;
```

---

## Explanation

*sqltype* is the integer code of the SQL type.

*sqllen* is the number of bytes in the data file for the specified SQL type.

## Note

The **rtypwidth** function is designed to be used with the **sqlda** structure that is returned by a DESCRIBE statement. *sqltype* and *sqllen* correspond to the components of the same name in each **sqlda.sqlvar** structure.

## Return Codes

0 *sqltype* is not a valid SQL type.

*n* (> 0) A value of type *sqltype* requires a minimum of *n* characters to be expressed.

# RUPSHIFT

## Overview

The **rupshift** function changes all of the characters within a null-terminated string to uppercase.

## Syntax

---

```
rupshift(s)  
char *s;
```

---

## Explanation

**s** is a pointer to a null-terminated string.

# SQLBREAK

## Overview

The **sqlbreak** function is a run-time SQL routine that sends the database engine a request to interrupt processing.

## Syntax

---

**sqlbreak( )**

---

## Note

When the database engine receives the interrupt signal, it returns status and control to the application process as if the SQL statement had terminated with an error condition.



# SQLEXIT

## Overview

The **sqlexit** function is a run-time SQL routine that terminates a database engine process, freeing resources. It can be used to reduce database overhead in programs that refer to a database only briefly and at long intervals or that access a database only during initialization.

## Syntax

---

**sqlexit()**

---

## Note

The **sqlexit** routine should be called only when there is no database open. For example, before calling **sqlexit**, issue a **CLOSE DATABASE** statement. If **sqlexit** is called when a database is open, it will roll back any current transactions and close the database.

# SQLSTART

## Overview

The **sqlstart** function is a run-time SQL routine that starts a database engine process.

## Syntax

---

**sqlstart()**

---

## Notes

1. The **sqlstart** routine should be called only when there is no database open. If it is called when a database is open, **sqlstart** does nothing.
2. Executing the **\$database** statement has the same effect as calling **sqlstart**, but also opens a database.

# STCAT

## Overview

The **stcat** function concatenates one null-terminated string to the end of another.

## Syntax

---

```
stcat(s, dest)  
char *s, *dest;
```

---

## Explanation

*s* is a pointer to the start of the string that is placed at the end of the destination string.

*dest* is a pointer to the start of the null-terminated destination string.

# STCHAR

## Overview

The **stchar** function stores a null-terminated string in a fixed-length string, padding the end with blanks if necessary.

## Syntax

---

```
stchar(from, to, count)
    char *from;
    char *to;
    int count;
```

---

## Explanation

<i>from</i>	is a pointer to the first byte of a null-terminated source string.
<i>to</i>	is a pointer to the fixed-length destination string.
<i>count</i>	is the number of bytes in the fixed-length destination string.



# STCMPR

## Overview

The **stcmp** routine compares two null-terminated strings.

## Syntax

---

```
stcmp(s1, s2)  
char *s1, *s2;
```

---

## Explanation

- s1* is a pointer to the first null-terminated string.
- s2* is a pointer to the second null-terminated string.

## Note

*s1* is greater than *s2* when *s1* appears after *s2* in the ASCII collating sequence.

## Return Codes

- =0 The two strings are identical.
- <0 The first string is less than the second.
- >0 The first string is greater than the second.

# STCOPY

## Overview

The **stcopy** function copies a null-terminated string from one location in memory to another location.

## Syntax

---

```
stcopy(from, to)  
      char *from, *to;
```

---

## Explanation

*from* is a pointer to the null-terminated string to be copied.

*to* is a pointer to a location in memory where the string is copied.

# STLENG

## Overview

The **stleng** function returns the length, in bytes, of a specified null-terminated string.

## Syntax

---

```
stleng(string)  
char *string;
```

---

## Explanation

*string* is a pointer to a null-terminated string.

## Note

The length does not include the terminating null.

## **Chapter 6**

### **C Functions in ACE and PERFORM**





## Chapter 6 Table of Contents

Chapter Overview .....	5
ACE Report Specification Files .....	6
Declaring C Functions .....	6
Calling C Functions .....	7
Compiling the Report Specification .....	9
PERFORM Form Specification Files.....	10
Calling the C Function.....	10
ON BEGINNING and ON ENDING.....	12
Compiling the Form Specification.....	13
Writing the C Program .....	13
C Program Structure .....	13
Input Parameters.....	16
Type Conversions.....	18
Return Values .....	18
Special PERFORM Library Functions .....	20
PF_GETTYPE.....	21
PF_GETVAL .....	23
PF_PUTVAL .....	25
PF_NXFIELD .....	27
PF_MSG.....	29
Compiling, Linking, and Running.....	30
Examples .....	31
ACE.....	32
Example 1.....	32
Example 2.....	34
PERFORM .....	36
Example 1.....	36
Example 2.....	39



# Chapter Overview

This chapter discusses interfacing C language routines with **INFORMIX-SQL**. If you have only **INFORMIX-ESQL/C**, this chapter is not relevant and can be skipped.

The **ACE** Report Writer and the **PERFORM** Screen Transaction Processor are the powerful report writing and interactive screen form processor programs of the Informix relational database management system, **INFORMIX-SQL**. Each of these programs uses a program (**ACEPREP** for **ACE** and **FORMBUILD** for **PERFORM**) to compile a user-created *specification file* that describes the customized report or screen form.

This chapter supplements the material in the ***INFORMIX-SQL Reference Manual*** by describing how you can call C functions from the specification file for both **ACE** and **PERFORM**. If your application uses **PERFORM**, this chapter assumes you are familiar with the material in Chapters 3 and 4 of the ***INFORMIX-SQL Reference Manual***. If your application uses **ACE**, it is assumed that you are familiar with Chapter 5 of the ***INFORMIX-SQL Reference Manual***.

While both **ACE** and **PERFORM** usually can handle all your database reports and screen interactions without modification, occasionally you may find it necessary to add a feature that is not present. For example, a C function called from **ACE** might make statistical computations on the data presented in a report and add these to the report. **PERFORM** might call C functions to check on the validity of data, to record the date, time, and name of the person updating the records, as well as to edit and to update the database.

C functions can use library routines described in this manual in Chapter 4, "SQL Data Types" and Chapter 5, "Library Functions" (and for **PERFORM**, the special functions at the end of this chapter). They can contain **INFORMIX-ESQL/C** statements and call math functions or other C subroutines. Your ability to call user-defined C functions increases the power and flexibility of **ACE** and **PERFORM**.

This chapter discusses how to call a C function from within the specification files for **ACEPREP** and **FORMBUILD**, how to construct the C function, and how to create custom versions of **ACE** and **PERFORM** containing your C function(s).



# ACE Report Specification Files

The general format of an ACE report specification file has seven sections:

---

DATABASE section (required)  
DEFINE section  
INPUT section  
OUTPUT section  
SELECT section (SELECT or READ required)  
READ section (SELECT or READ required)  
FORMAT section (required)

---

You call a C function from within a report specification file by declaring the function name in the DEFINE section and by using the function in the FORMAT section. Then use ACEPREP to compile the report specification file.

## Declaring C Functions

You declare a C function in the DEFINE section of the report specification file.

### Syntax

---

```
DEFINE  
    FUNCTION userfunc  
END
```

---

### Explanation

DEFINE            is a required keyword.

FUNCTION        is a required keyword.

*userfunc*

is the name by which the C function is referenced in the specification file. *userfunc* must satisfy the conditions for an ACE identifier.

END

is a required keyword.

## Notes

1. You can declare several functions at the same time by repeating the keyword FUNCTION followed by the next function name.
2. Do not include parentheses after the function name.
3. You can have PARAM, VARIABLE, and ASCII statements within the DEFINE section in addition to the FUNCTION statement.

## Calling C Functions

The FORMAT section of the report specification file contains one or more *control blocks* that determine when ACE will take an action.

---

### FORMAT

PAGE HEADER control block  
PAGE TRAILER control block  
FIRST PAGE HEADER control block  
ON EVERY ROW control block  
ON LAST ROW control block  
BEFORE GROUP OF control block  
AFTER GROUP OF control block

END

---

Each control block contains one or more *statements* that tell ACE what action to take. Chapter 5 of the *INFORMIX-SQL Reference Manual* describes a variety of statements that ACE allows. In addition to the statements defined there, you can use a C function call.

---

[CALL] *userfunc*([*expr-list*])

---

### Explanation

- CALL** is an optional keyword that must be used when the C function does not return a value. When CALL is missing, *userfunc* must return a value.
- userfunc* is the name of a C function that has been previously declared in the DEFINE section.
- expr-list* is 1 to 10 expressions separated by commas.

### Notes

1. An expression can be anything from a simple numeric or alphabetic constant to a more complex series of column names, ACE variables, ACE parameters, ACE functions (such as group aggregates and date functions), quoted strings, arithmetic and logical operators, and keywords.
2. ACE statements are composed of keywords and expressions. You also can use a C function in an expression where you can use a constant. In this case, do not include the CALL keyword, and the C function must return a value.

## Examples

---

```
after group of order_num  
call stat(order_num)
```

---

This control block calls a C function `stat` that calculates statistics on the data in the rows that correspond to the order number = `order_num`.

---

```
on every row  
  print order_num,  
    logarithm((total of total_price)/(today - order_date))
```

---

This control block prints the name and a value intended to correlate the total price of each order with the period of time the order has been outstanding. It calls a C function that computes the logarithm.

---

```
first page header  
call to_unix("date")
```

---

This control block is taken from ACE Example 1 at the end of this chapter. It prints the system date and time at the top of the first page of the report. The function `to_unix` sends its string argument to the UNIX operating system.

## Compiling the Report Specification

Use **ACEPREP** to compile the report specification when there are C function calls, just as you would when there are none. Give the report specification file a filename with the extension **.ace**, such as **specfile.ace**. To invoke **ACEPREP** for this file, enter the following statement on the command line:

```
saceprep specfile
```

See Chapter 5 of the *INFORMIX-SQL Reference Manual* for further details.



# PERFORM Form Specification Files

The general format of a **PERFORM** form specification file has five sections:

---

DATABASE section (required)  
SCREEN section (required)  
TABLES section (required)  
ATTRIBUTES section (required)  
INSTRUCTIONS section

---

You can use C functions in the control blocks in the **INSTRUCTIONS** section of a form specification file.

## *Calling the C Function*

In control blocks, you can use C functions anywhere you can use an expression, or they can stand alone as the most general action. The syntax for a call to a user-defined function is as follows:

### Syntax

---

[CALL] *userfunc*(*expr-list*)

---

### Explanation

**CALL** is an optional keyword that must be used when the C function does not return a value. When **CALL** is missing, *userfunc* must return a value.

*userfunc* is the name by which the C function is referenced in the **PERFORM** specification.

*expr-list* is a list of 0 to 10 expressions. An expression is defined as:

- a field tag
- a constant value
- an aggregate value
- a C function
- the keyword TODAY
- the keyword CURRENT
- any combination of the above, combined by using the arithmetic operators +, -, \*, and /

## Examples

---

```
after editadd of proj_num
  let f001 = userfunc(f002)

before editupdate of paid_date
  if boolfunc(f003) then
    let f004 = 15
  else
    let f004 = 10

after add update remove of customer
  call userfunc( )
```

---

For more information about the LET and IF-THEN-ELSE actions and expressions, refer to Chapter 3 of the *INFORMIX-SQL Reference Manual*.

## ***ON BEGINNING and ON ENDING***

ON BEGINNING and ON ENDING are two control blocks that can be used only with calls to C functions. These control blocks are specified in the INSTRUCTIONS section of the form specification file. ON BEGINNING executes immediately after PERFORM is invoked, and ON ENDING executes immediately after the EXIT command.

- By using ON BEGINNING, you can give instructions, request a special password, or initialize a temporary workfile in which to keep a batch of transaction records.
- By using ON ENDING, you can perform calculations to summarize the changes made in the database during the PERFORM session that just concluded, print summaries of the records added, or erase workfiles.

You can have more than one ON BEGINNING and/or ON ENDING control block in the INSTRUCTIONS section. However, you can have only one CALL statement in each control block.

### **Syntax**

---

ON BEGINNING  
    CALL *userfunc(expr-list)*

ON ENDING  
    CALL *userfunc(expr-list)*

---

The explanation of the syntax is identical to that in the preceding section.

## ***Compiling the Form Specification***

Use **FORMBUILD** to compile the form specification file when there are C function calls in the same manner as when there are none. Give the form specification file a filename with the extension **.per**, such as **specfile.per**. To invoke **FORMBUILD** with this file, enter the following statement on the command line:

```
sformblld specfile
```

See Chapter 3 of the *INFORMIX-SQL Reference Manual* for further details.

## **Writing the C Program**

If you make reference to C functions within an **ACE** report specification file or a **PERFORM** form specification file, you must create a custom version of **ACE** or **PERFORM** that knows the functions. You must write a C program that includes the appropriate header files and structure declarations, as well as your functions. You then must compile and link your program to the appropriate libraries. This section shows you how to organize your C program, how to declare your functions, and how to pass values to and from your functions.

### ***C Program Structure***

To create a version of **ACE** or **PERFORM** that includes your functions, you must write a C program that contains the appropriate declarations. Your program can have one or more functions, and you can define other functions to use internally in your program.

The following display shows the general structure of such a C program that includes two user-defined functions:



---

```

#include "ctools.h"
/* add other includes as desired */

valueptr funct1();
valueptr funct2();

struct ufunc userfuncs[] =
{
    "myfunct1", funct1,
    "myfunct2", funct2,
    0,0
};

/* add other global declarations */

valueptr funct1()
{
    .
    .
    .
    /* funct1 takes no arguments
       and returns a character string */
    .
    .
    .
    strreturn(s, len);
}

valueptr funct2(arg1, arg2)
valueptr arg1, arg2;
{
    .
    .
    .
    /* funct2 takes two arguments
       and returns no value */
    .
    .
    .
}

```

---

A description of the structure of the preceding program follows.

1. At the top of your C program, you must have the following line:

```
#include "ctools.h"
```

See the **ctools.h** header file in Appendix A.

You may want to include other files, such as **math.h** or **stdio.h**, depending on your application. If you use **INFORMIX-ESQL/C**, you can include **sqlca.h** and other header files.

2. Before you initialize the required array of **ufunc** structures, you must declare your functions. Included in **ctools.h** is the definition of the **value** structure and pointers to that structure.

---

```
typedef struct value *valueptr;  
typedef struct value *acevalue;  
typedef struct value *perfvalue;
```

---

The last two pointers are included for compatibility with earlier versions of **INFORMIX**. All of your functions must be of type **valueptr**. If **funct1( )** and **funct2(arg1, arg2)** are your functions, their declarations must come next.

---

```
valueptr funct1( );  
valueptr funct2( );
```

---

If you place the initialization of the **userfuncs** array after the code that defines your functions, you may skip this step.

3. Make the structure declaration and initialization for **userfuncs[ ]** the next section of your program. These structures are required so that **ACE** and **PERFORM** can call your functions at run time.

---

```
struct ufunc userfuncs[] =
{
    "myfunct1", funct1,
    "myfunct2", funct2,
    0,0
};
```

---

The quoted strings, "**myfunct1**" and "**myfunct2**", must be the names of the functions as they are used in the specification file. **funct1** and **funct2**, which correspond to "**myfunct1**" and "**myfunct2**", respectively, are pointers to the functions as defined within the C program. Note that the C functions as defined here need not have the same name that you used in your specification file. The purpose of the **userfuncs** array is to make the connection between these two names. The two zeros at the end of the array are required as terminators.

4. The last section of the C program is the code for your functions. As stated earlier, all of the functions that you call in **ACE** or **PERFORM** must be declared as returning pointers to a **value** structure. Also, all arguments of your functions must be declared of type **valueptr**.

The call to **strreturn** in the definition of **funct1** is an example of the use of one of several macros that can be used to return values of type **valueptr**. These and other conversion routines are described in a later section.

## *Input Parameters*

The **ctools.h** header file is designed to make the process of interfacing C functions to **ACE** and **PERFORM** easy. For example, if the parameter passed to the C function is **arg**, with **INFORMIX-SE**, the following definitions can be used to detect the data type of **arg** and to extract the value of **arg**, no matter what its data type.

---

<code>arg-&gt;v_charp</code>	pointer to string
<code>arg-&gt;v_len</code>	length of string
<code>arg-&gt;v_int</code>	integer value
<code>arg-&gt;v_long</code>	long value
<code>arg-&gt;v_float</code>	float value
<code>arg-&gt;v_double</code>	double value
<code>arg-&gt;v_decimal</code>	decimal, money, datetime, or interval value
<code>arg-&gt;v_type</code>	data type
<code>arg-&gt;v_ind</code>	null indicator
<code>arg-&gt;v_prec</code>	datetime/interval qualifier

---

The data type of `arg` can be determined by checking `arg->v_type` against a series of integer constants defined in `ctools.h`.

---

<code>v_type</code>	SQL Type	C Type
SQLCHAR	CHAR	{ char string fixchar
SQLSMINT	SMALLINT	short
SQLINT	INTEGER	long
SQLFLOAT	FLOAT	double
SQLSMFLOAT	SMALLFLOAT	float
SQLDECIMAL	DECIMAL	dec_t
SQLSERIAL	SERIAL	long
SQLDATE	DATE	long
SQLMONEY	MONEY	dec_t
SQLDTIME	DATETIME	dtime_t
SQLINTERVAL	INTERVAL	intrvl_t

---

If `arg->v_type` is `SQLCHAR`, then the pointer to the string is available in `arg->v_charp`, and the number of characters in the string (length) is available in `arg->v_len`. The string is *not* null-terminated.

`arg->v_ind` is negative if the value of `arg` is `NULL`, and zero otherwise.

---

**INFORMIX-OnLine** supports additional data types. Refer to the *INFORMIX-OnLine Programmer's Manual* for more information.

---



## Type Conversions

The **ctools.h** header file provides an alternative to testing the type of the parameter passed from **ACE** or **PERFORM**. Several functions, listed below, can force conversion of a parameter passed as a pointer to a **value** structure, to a C data type of your choice:

---

Function	Returned Type
<b>toint</b>	<b>int</b>
<b>tolong</b>	<b>long</b>
<b>tofloat</b>	<b>double</b>
<b>todouble</b>	<b>double</b>
<b>todate</b>	<b>long</b>
<b>todecimal</b>	<b>dec_t</b>
<b>todatetime</b>	<b>dtime_t</b>
<b>tointerval</b>	<b>intrvl_t</b>

---

These functions take a pointer to a type **value** structure and return a value of the type indicated. **todecimal** takes a second argument that is a pointer to the **dec\_t** structure. **todatetime** takes a second argument that is a pointer to the **dtime\_t** structure. **tointerval** takes a second argument that is a pointer to the **intrvl\_t** structure. If the type conversion is not successful, the global integer **toerrno** will be set to a negative value; **toerrno** is zero if the conversion is successful.

## Return Values

When your function is expected to return a value to **ACE** or **PERFORM**, you must insert the value in a type **value** structure and return a pointer to that structure. **ctools.h** contains the following macros to perform that procedure for you:

---

Macro	Type Returned
intreturn(i)	returns integer i
lngreturn(l)	returns long l
floreturn(f)	returns float f
dubreturn(d)	returns double d
strreturn(s,c)	returns string s of length c (short)
decreturn(d)	returns decimal d (of type dec_t)
dtimereturn(d)	returns datetime d (of type dtime_t)
invreturn(i)	returns interval i (of type intrvl_t)

---

Use the appropriate macro even when you want to return an error condition. Do not use a simple **return**.

Since **strreturn(s,c)** returns a pointer to the string **s**, be sure to define **s** as a static or external variable.

---

**INFORMIX-OnLine** supports additional data types. Refer to the *INFORMIX-OnLine Programmer's Manual* for more information.

---

# Special PERFORM Library Functions

Five C functions are designed to control **PERFORM** screens from within C functions:

<b>pf_gettype</b>	determines the type and length of a display field on a screen.
<b>pf_getval</b>	reads a value from a display field.
<b>pf_putval</b>	puts a value onto a display field.
<b>pf_nxfield</b>	moves the cursor to a specified field.
<b>pf_msg</b>	writes a message at the bottom of the screen.

These functions are described in detail on the following pages. Their return value is zero if successful, or a non-zero error code if unsuccessful.

**Note:** The argument of **pf\_getval** and **pf\_putval** that refers to the value returned or displayed must be a pointer to the variable containing the value. A common programming error is to use the variable itself. This results in a run-time system error and is not detected by the compiler.

# PF\_GETTYPE

## Overview

**pf\_gettype** returns the SQL data type and the length of the display field for a specified field tag.

## Syntax

---

```
pf_gettype(tagname, type, len)
    char *tagname;
    short *type, *len;
```

---

## Explanation

*tagname* is a string containing the field tag that specifies a display field.

*type* is a pointer to a short integer that describes the data type of the display field *tagname*.

*len* is a pointer to a short integer that is the length of the display field *tagname* on the **PERFORM** screen.



## Note

These are the possible values of *type*. They are defined in **ctools.h**:

---

<b>type</b>	<b>SQL Type</b>
SQLCHAR	CHAR
SQLSMINT	SMALLINT
SQLINT	INTEGER
SQLFLOAT	FLOAT
SQLSMFLOAT	SMALLFLOAT
SQLDECIMAL	DECIMAL
SQLSERIAL	SERIAL
SQLDATE	DATE
SQLMONEY	MONEY
SQLDTIME	DATETIME
SQLINTERVAL	INTERVAL

---

## Return Codes

0	The operation was successful; display field was found.
3759	There is no such field tag in the form.

# PF\_GETVAL

## Overview

**pf\_getval** obtains the value found in a display field and its length, if it is a character field.

## Syntax

---

```
pf_getval(tagname, retvalue, valtype, vallen)
    char *tagname, *retvalue;
    short valtype, vallen;
```

---

## Explanation

- tagname* is a string containing the field tag that specifies a display field.
- retvalue* is a pointer to the string, short, long, float, double, decimal, datetime, or interval structure returned by **pf\_getval**.
- valtype* is a short integer indicating the type of value to which *retvalue* should point.
- vallen* is a short integer specifying the length of the string (plus 1 for the terminating null byte) returned in *retvalue*, when *valtype* is CCHARTYPE. For any other value for *valtype*, *vallen* is ignored.

---

INFORMIX-OnLine supports additional data types. Refer to the *INFORMIX-OnLine Programmer's Manual* for more information.

---

## Notes

1. The options for *valtype* are as follows:

---

<b>valtype</b>	<b>SQL Type</b>
CCHARTYPE CFIXCHARTYPE CSTRINGTYPE CINTTYPE CSHORTTYPE CLONGTYPE CFLOATTYPE CDOUBLETYPE CDECIMALTYPE CDATETIME CINTERVAL	CHAR  INTEGER SMALLINT INTEGER, DATE, SERIAL SMALLFLOAT FLOAT DECIMAL, MONEY DATETIME INTERVAL

---

2. The type of *retvalue* is determined by the value given to the parameter *valtype*. *valtype* need not correspond exactly to the data type of the display field, but both should be either a number or character so that **PERFORM** can do the proper type conversion.
3. If *valtype* is a number field and the display field is a character field, type conversion is done, if possible. When the conversion cannot be completed successfully, the number pointed to by *retvalue* is zero. If *valtype* is character and the display field is a number field, a conversion to a string occurs. If the string does not fit in the length specified by *vallen*, *retvalue* contains the string, truncated to fit and null-terminated.

## Return Codes

0	The operation was successful; display field was found.
3700	The user is not permitted to read the field.
3759	There is no such field tag in the form.

# PF\_PUTVAL

## Overview

**pf\_putval** puts a value into a **PERFORM** screen in a specified display field. The user must have permission to update or to enter data into the desired destination field.

## Syntax

---

```
pf_putval(pvalue, valtype, tagname)
    char *pvalue;
    short valtype;
    char *tagname;
```

---

## Explanation

- pvalue* is a pointer to a string, short, integer, long, float, double, decimal, datetime, or interval structure inserted into the display field designated by *tagname*.
- valtype* is a short integer indicating the type of value to which *pvalue* points.
- tagname* is a string containing the field tag specifying the display field where the information pointed to by *pvalue* is placed.

---

**INFORMIX-OnLine** supports additional data types. Refer to the *INFORMIX-OnLine Programmer's Manual* for more information.

---



## Notes

1. The options for *valtype* are as follows:

---

valtype	SQL Type
CCHARTYPE CFIXCHARTYPE CSTRINGTYPE	CHAR
CINTTYPE	
CSHORTTYPE	
CLONGTYPE	INTEGER
CFLOATTYPE	SMALLINT
CDOUBLETYPE	INTEGER, DATE
CDECIMALTYPE	SMALLFLOAT
CDATETIME	FLOAT
CINTERVAL	DECIMAL, MONEY
	DATETIME
	INTERVAL

---

2. The data type of *pvalue* is determined by the value given to the parameter *valtype*.
3. If *valtype* is one of the character types and the display field is a number field, type conversion is done, if possible. If the conversion cannot be completed successfully, the value 0 is entered into the display field.
4. If the type specified is a number field and the display field is character, a conversion to a string occurs. If the string does not fit in the display field, the display field is truncated.
5. If a number value does not fit in a number display field, the field is filled with asterisks.

## Return Codes

0	The operation was successful; display field was found.
3710	The user is not permitted to update the field.
3720	The user is not permitted to add to the field.
3756	The display field is not in the current table.
3759	There is no such field tag in the form.

# PF\_NXFIELD

## Overview

**pf\_nxfield** controls the cursor placement on a **PERFORM** screen when in an edit mode (either adding a new record or updating an old one).

## Syntax

---

```
pf_nxfield(tagname)  
char *tagname;
```

---

## Explanation

*tagname* is a string containing the field tag for the display field on a **PERFORM** screen to which the cursor is sent.

## Notes

1. If **pf\_nxfield** is called during a **BEFORE EDITADD** or a **BEFORE EDITUPDATE** of a table, it controls which display field is edited first.
2. If **pf\_nxfield** is called during an **AFTER EDITADD** or an **AFTER EDITUPDATE** of a table, it causes the cursor to move to the designated display field *tagname* for further editing, rather than allowing the record to be written.
3. If **pf\_nxfield** is called either **BEFORE** or **AFTER** an **EDITADD** or **EDITUPDATE** of a column, it determines the next field to be edited.
4. If **pf\_nxfield** is called either **AFTER ADD** or **AFTER UPDATE**, it is inoperative, since the record has already been written.

5. If *tagname* is set equal to the value EXITNOW, **pf\_nxfield** causes an immediate exit from the add or update operation with the row being added or updated. This option performs the same as when you press ESCAPE to complete the transaction.

## Return Codes

0	The operation was successful; display field was found.
3710	The user is not permitted to update the field.
3720	The user is not permitted to add to the field.
3755	The display field is display-only.
3756	The display field is not in the current table.
3759	There is no such field tag in the form.

# PF\_MSG

## Overview

**pf\_msg** displays a message at the bottom of the screen.

## Syntax

---

```
pf_msg(msgstr, reverseflag, bellflag)  
    char *msgstr;  
    short reverseflag, bellflag;
```

---

## Explanation

<i>msgstr</i>	is a string containing the message displayed at the bottom of the screen.
<i>reverseflag</i>	is a short integer indicating whether the message is displayed in reverse video (0 = normal video, 1 = reverse video).
<i>bellflag</i>	is a short integer indicating whether the terminal bell is rung when the message is displayed (0 = no bell, 1 = bell).

## Notes

1. If several calls to **pf\_msg** are invoked at the same time in response to satisfying several conditions simultaneously, only the last message displayed is visible to the user.
2. In normal video display, the string can have up to 80 characters. In reverse video display, the maximum number of characters is less than 80 because the reverse video control characters require one or more spaces on some terminals.



# Compiling, Linking, and Running

After you have written the file containing your C functions, you must compile them and link the necessary library functions to create a custom version of **sacego** or **sperform**.

After you compile your custom version of **sacego** or **sperform**, you can run reports or forms with the following command line

```
custprog specfile
```

where *specfile* is the name of the report or form specification file you compiled using **ACEPREP** or **FORMBUILD**.

Shell scripts are provided to simplify the compiling and linking process. You can use these shell scripts as you would the standard C compiler-linker **cc**. You need not be concerned with names of special **ACE**, **PERFORM**, or **INFORMIX-ESQL/C** libraries or preprocessors, nor with the location of the include files associated with these programs.

## Syntax

---

```
[cace | cperf] cprogram.[c | ec] [...] -o custprog other-C-list
```

---

## Explanation

<b>cace</b>	is the shell script that creates a custom version of <b>sacego</b> .
<b>cperf</b>	is the shell script that creates a custom version of <b>sperform</b> .
<b>cprogram</b>	is the name of the C program that contains your functions, as described in the previous sections.
<b>.c</b>	is the extension to use if <b>cprogram</b> has no <b>INFORMIX-ESQL/C</b> statements.

<b>.ec</b>	is the extension to use if <b>cprogram</b> has <b>INFORMIX-ESQL/C</b> statements (see Chapter 1, "Using SQL").
<b>-o</b>	specifies the output filename.
<b>custprog</b>	is the name of your custom version of <b>sacego</b> or <b>sperform</b> .
<b>other-C-list</b>	is the rest of the arguments that you want to pass to the standard <b>cc</b> program.

### **Note**

You can compile several C programs at the same time.

## **Examples**

This section contains examples of both **ACE** applications and **PERFORM** applications. **ACE** C functions can be used with **PERFORM** as well. These sample programs are delivered with the demonstration database. However, you also must have **INFORMIX-SQL** to compile and execute these programs.

## Example 1

Figure 6-1 shows a specification file that calls a user function to execute a system command. The program is **a\_ex1.ace** in the demonstration database. The user function in **to\_unix.c** is shown in Figure 6-2.

---

```
database
    stores
end

define
    function to_unix
end

select * from customer
end

format
    first page header
        call to_unix("date")
        skip 1 line
    on every row
        print customer_num, 3 spaces,
            fname clipped, 1 space, lname
end
```

---

**Figure 6-1.** ACE Example 1: Specification File

---

```
#include "ctools.h"

valueptr to_unix( );

struct ufunc userfuncs[] =
{
    "to_unix", to_unix,
    0,0
};

valueptr to_unix(string)
valueptr string;
{
    char savearea[80];

    /*copy bytes from string to savearea*/
    bycopy(string->v_charp, savearea, string->v_len);

    /*put null on end*/
    savearea[string->v_len]=0;

    system(savearea);
}
```

---

**Figure 6-2.** ACE Example 1: *to\_unix* Function



## Example 2

Figure 6-3 shows an ACE program calling a C function that computes the square root of a DECIMAL type number. The C function uses some of the decimal functions described in Chapter 4, "SQL Data Types."

This program, `a_ex2.ace` in the demonstration database, computes the average and the standard deviation of the total cost of all the orders in the `stores` database. The user function in `decsqrt.c` is shown in Figure 6-4.

---

```
database
  stores
end

define
  function decsqroot
end

select o.order_num, sum(total_price) t_cost
  from orders o, items i
  where o.order_num = i.order_num
  group by o.order_num
end

format
  on every row
    print order_num, t_cost
  on last row
    skip 1 line
    print "The average total order is      : ",
      (total of t_cost)/count
      using "$#####.##"
    print "Standard deviation is          : ",
      decsqroot((total of t_cost*t_cost)/count
        - ((total of t_cost)/count)**2)
      using "$#####.##"
end
```

---

**Figure 6-3.** ACE Example 2: Specification File

---

```
#include "ctools.h"
#include <math.h>

valueptr squareroot( );

struct ufunc userfuncs[] =
{
    "decsqroot", squareroot,
    0, 0
};

valueptr squareroot(pnum)
valueptr pnum;
{
    double dub;
    dec_t dec;

    /* convert decimal to double */
    dectodbl(&pnum->v_decimal, &dub);

    dub = sqrt(dub);

    /* convert double to decimal */
    deccvdbl(dub, &dec);

    /* return decimal */
    decreturn(dec);
}
```

---

**Figure 6-4.** ACE Example 2: *decsqroot* Function

# PERFORM

## Example 1

It is often useful in database applications to record the identification of the entry clerk and the time of data entry with the data entered. It is not necessary to require the entry clerk to enter these items. The UNIX operating system can identify the entry clerk through the login name and also can supply the time. Using the techniques described in this chapter, you can write a C program to gather these items from the operating system and display them on the screen. **PERFORM** adds them to the row when the row is written.

Figure 6-5 illustrates a form specification file for entering new customers into the **stores** database. The form is in **p\_ex1.per** in the demonstration database; **stamp.c** contains the function **stamptime**.

Assume for the purpose of this example that the **customer** table has two additional columns: **entry\_clerk** and **entry\_time**, both of type **CHAR(10)**. The form automatically records the login name of the entry clerk and the time that the customer is added to the database.

The cursor moves from the upper left down through the Customer Data by following the order of the fields listed in the **ATTRIBUTES** section. After the Telephone field, the cursor moves to the Owner Name field. When the entry clerk presses **ESCAPE** to complete the transaction, the C function **stamptime** is called.

```

database stores
screen
{
    *****
    *                               Customer Form                               *
    *=====
    * Number      :[f000          ]                                           *
    * Owner Name  :[f001          ][f002          ]                             *
    * Company     :[f003          ]                                           *
    * Address     :[f004          ]                                           *
    *             :[f005          ]                                           *
    * City        :[f006          ] State:[a0] Zipcode:[f007 ]                 *
    * Telephone   :[f008          ]                                           *
    *=====
    * Entry Clerk :[f009          ]      Time Entered :[f010          ] *
    *****
}

tables
    customer

attributes
f000 = customer.customer_num, noentry;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.company;
f004 = customer.address1;
f005 = customer.address2;
f006 = customer.city;
a0 = customer.state, default="CA", upshift, autonext;
f007 = customer.zipcode, autonext;
f008 = customer.phone;
f009 = customer.entry_clerk;
f010 = customer.entry_time;

instructions

after editadd editupdate of phone
    nextfield = f001

after editadd editupdate of customer
    call stamptime( )

end

```

**Figure 6-5. PERFORM Example 1: Specification File**

The function **stamptime**, called by the form specification file when the entry clerk presses **ESCAPE** to complete the transaction, is shown in Figure 6-6. In addition to the special function **pf\_putval** defined earlier in this chapter, **stamptime** uses the system functions **time**, **localtime**, and **getlogin**.



The login name of the order taker is obtained from the string function **getlogin** and displayed in the screen field Entry Clerk.

The system time is decomposed into hours and minutes and then reconstructed into a string variable put in the screen field Time Entered. **PERFORM** then writes the record to the **customer** table, using the data on the screen.

---

```
#include <stdio.h>
#include <time.h>
#include "ctools.h"

valueptr stamptime( );

struct ufunc userfuncs[] =
{
    "stamptime", stamptime,
    0,0
};

valueptr stamptime( )
{
    long seconds, time( );
    char usertime[10], *getlogin( );
    struct tm *timerec, *localtime( );

    seconds = time((long *) 0);
    timerec = localtime(&seconds);

    pf_putval(getlogin( ), CCHARTYPE, "f009");

    sprintf(usertime, "%02d:%02d",
        timerec->tm_hour, timerec->tm_min);
    pf_putval(usertime, CCHARTYPE, "f010");
}
```

---

**Figure 6-6.** PERFORM Example 1: *stamptime* Function

## Example 2

Ordinarily, you can enter data into only a single row at a time on a **PERFORM** screen. Using the display-only fields allowing input, control blocks, and the C function calls described here, you can create a display-only screen into which data can be entered for several rows at once. After you enter the data and press **ESCAPE**, your C program interrogates the screen and writes the actual rows into the tables.

Figure 6-7 shows an order form. It has room for listing up to five items with the quantity and total price. The extension to more items is clear both in the entry form and in the C program that follows. Note that all the display fields are either display-only fields or are *lookup* fields based upon the values entered into those display-only fields that permit data entry.

Upon entering the **PERFORM** program, a function call is made to open the **stores** database.

To use the form, you must have a list of customer numbers (**customer\_num**) for all existing customers. After you enter the customer number into display field **f000**, **PERFORM** displays the customer's first and last names and address in the fields **f001** through **f007**. The cursor then moves to field **f010** (Order Date) where you can begin entering the order description. Since the default for the Order Date is **today**, this field appears already filled. You can type in an alternative date.

After you enter data in the fields **f011** and **f012**, the cursor moves to the list of items and you enter the stock number (**stock\_num**) and manufacturer code (**manu\_code**) for the first item. **PERFORM** displays the description of the stock item selected and the unit price. It calls the function **st\_desc** to perform a lookup in the **stock** table and leaves the cursor in the field **q1**, waiting for you to enter a quantity.

When you have entered a quantity, **PERFORM** computes the total price for that item and keeps a running total of all items at the bottom of the screen. The cursor then moves to the first column and waits for the stock number of the second item.

The process continues through the list of items until you press **ESCAPE** and complete the transaction. **PERFORM** then enters the order data into the **orders** table and each of the items into the **items** table. It also enters the new Order Number on the screen.

When you leave **PERFORM**, a call is made to a function to close the database.

database stores

screen

{

-----  
ORDER FORM  
-----

Customer Number:[f000           ]   Contact Name:[f001           ] [f002  
Company Name:[f003               ]                                    ]  
Address:[f004                    ] [f005                            ]  
City:[f006                        ] State:[a0] Zip Code:[f007 ]

-----  
Order No:[f009           ]   Order Date:[f010           ]   Purchase Order No:[f011  
Shipping Instructions:[f012    ]  
-----

Stock No.	Code	Description	Quantity	Price	Total
[sn1 ]	[mc1]	[de1                   ]	[q1 ]	[pr1 ]	[tp1 ]
[sn2 ]	[mc2]	[de2                   ]	[q2 ]	[pr2 ]	[tp2 ]
[sn3 ]	[mc3]	[de3                   ]	[q3 ]	[pr3 ]	[tp3 ]
[sn4 ]	[mc4]	[de4                   ]	[q4 ]	[pr4 ]	[tp4 ]
[sn5 ]	[mc5]	[de5                   ]	[q5 ]	[pr5 ]	[tp5 ]
Running Total including Tax and Shipping Charges:[rt					]

}

END

tables

customer  
orders  
items  
stock

attributes

f000 = displayonly allowing input type integer,  
lookup f001 = customer.fname,  
f002 = customer.lname,  
f003 = customer.company,  
f004 = customer.address1,  
f005 = customer.address2,  
f006 = customer.city,  
a0 = customer.state,  
f007 = customer.zipcode  
joining \*customer.customer\_num;  
f009 = displayonly type integer;  
f010 = displayonly allowing input type date, default = today;  
f011 = displayonly allowing input type char;  
f012 = displayonly allowing input type char;  
  
sn1 = displayonly allowing input type smallint;  
mc1 = displayonly allowing input type char, upshift;  
q1 = displayonly allowing input type smallint;  
sn2 = displayonly allowing input type smallint;



```

mc2 = displayonly allowing input type char, upshift;
q2  = displayonly allowing input type smallint;
sn3 = displayonly allowing input type smallint;
mc3 = displayonly allowing input type char, upshift;
q3  = displayonly allowing input type smallint;
sn4 = displayonly allowing input type smallint;
mc4 = displayonly allowing input type char, upshift;
q4  = displayonly allowing input type smallint;
sn5 = displayonly allowing input type smallint;
mc5 = displayonly allowing input type char, upshift;
q5  = displayonly allowing input type smallint;

```

```

del = displayonly type char;
pr1 = displayonly type money, right;
tp1 = displayonly type money, right;
de2 = displayonly type char;
pr2 = displayonly type money, right;
tp2 = displayonly type money, right;
de3 = displayonly type char;
pr3 = displayonly type money, right;
tp3 = displayonly type money, right;
de4 = displayonly type char;
pr4 = displayonly type money, right;
tp4 = displayonly type money, right;
de5 = displayonly type char;
pr5 = displayonly type money, right;
tp5 = displayonly type money, right;
rt  = displayonly type money, reverse, right;

```

instructions

```

after editadd of mc1
  if st_desc(1) then
    nextfield = q1
  else
    begin
      comments "Not a valid stock item"
      let sn1 = null
      let mc1 = null
      nextfield = sn1
    end
after editadd of mc2
  if st_desc(2) then
    nextfield = q2
  else
    begin
      comments "Not a valid stock item"
      let sn2 = null
      let mc2 = null
      nextfield = sn2
    end
after editadd of mc3
  if st_desc(3) then
    nextfield = q3
  else
    begin
      comments "Not a valid stock item"
      let sn3 = null
      let mc3 = null
      nextfield = sn3
    end

```



```

after editadd of mc4
  if st_desc(4) then
    nextfield = q4
  else
    begin
      comments "Not a valid stock item"
      let sn4 = null
      let mc4 = null
      nextfield = sn4
    end
after editadd of mc5
  if st_desc(5) then
    nextfield = q5
  else
    begin
      comments "Not a valid stock item"
      let sn5 = null
      let mc5 = null
      nextfield = sn5
    end
after editadd of q1
  let tp1 = q1 * pr1
  let rt = tp1 * 1.1
after editadd of q2
  let tp2 = q2 * pr2
  let rt = (tp1 + tp2) * 1.1
after editadd of q3
  let tp3 = q3 * pr3
  let rt = (tp1 + tp2 + tp3) * 1.1
after editadd of q4
  let tp4 = q4 * pr4
  let rt = (tp1 + tp2 + tp3 + tp4) * 1.1
after editadd of q5
  let tp5 = q5 * pr5
  let rt = (tp1 + tp2 + tp3 + tp4 + tp5) * 1.1
before editadd of displaytable
  let de1 = null
  let de2 = null
  let de3 = null
  let de4 = null
  let de5 = null
  let pr1 = null
  let pr2 = null
  let pr3 = null
  let pr4 = null
  let pr5 = null
  let tp1 = null
  let tp2 = null
  let tp3 = null
  let tp4 = null
  let tp5 = null
  let rt = null
after editadd of displaytable
  let f009 = add_order()
end

```

---

**Figure 6-7. PERFORM Example 2: Specification File**

**PERFORM** calls **st\_desc** when you leave the **Code** field for each item, and **add\_order** when you press **ESCAPE**. These functions are shown in Figure 6-8. In addition to the C functions defined in this chapter, the program uses some functions from Chapter 5, "Library Functions," as well as **INFORMIX-ESQL/C**.

The **st\_desc** function gets the values that you entered for **stock\_num** and **manu\_code** and checks whether either is **NULL**. If neither is **NULL**, it selects the **description** and **unit\_price** for the stock item defined by **stock\_num** and **manu\_code**, displays them on the screen, and returns the value 1 (true). If either of **stock\_num** or **manu\_code** is **NULL** or if there is no item in the **stock** table corresponding to them, **st\_desc** returns 0 (false).

The **add\_order** function begins by checking whether any items have been entered. If not, it writes a message saying that no order has been written and returns. If at least one item has been entered, **add\_order** collects the information about the order and starts a transaction. After inserting a new row in the **orders** table, **add\_order** enters a loop to insert each item of the order into the **items** table. It determines whether to insert a row into the **items** table by testing to see if the total price for that row is not **NULL**. Only if all of the changes to the database corresponding to the order are made successfully, is the work committed and the order actually entered. **add\_order** returns the value of the **order\_num** for **PERFORM** to display on the screen.

The form is in **p\_ex2.per** in the demonstration database, while **mult\_item.ec** contains the C functions.

```

#include <ctools.h>
#include sqlca;

extern valueptr st_desc();
extern valueptr add_order();

struct ufunc userfuncs[] =
{
    "st_desc", st_desc,
    "add_order", add_order,
    0, 0
};

char *sn[] = {
    "sn1",
    "sn2",
    "sn3",
    "sn4",
    "sn5"
};

char *mc[] = {
    "mc1",
    "mc2",
    "mc3",
    "mc4",
    "mc5"
};

char *de[] = {
    "de1",
    "de2",
    "de3",
    "de4",
    "de5"
};

char *q[] = {
    "q1",
    "q2",
    "q3",
    "q4",
    "q5"
};

char *pr[] = {
    "pr1",
    "pr2",
    "pr3",
    "pr4",
    "pr5"
};

char *tp[] = {
    "tp1",
    "tp2",
    "tp3",
    "tp4",
    "tp5"
};

```

```

valueptr st_desc(item)
valueptr item;
{
$   int s;
$   char m[4];
$   char d[16];
$   dec_t   up;
$   int i;

    i = item->v_int - 1;
    pf_getval(sn[i], &s, CINTTYPE, 0);
    pf_getval(mc[i], m, CCHARTYPE, 4);
    if (risnull(CINTTYPE, &s) || risnull(CCHARTYPE, m))
        intreturn(0); /* return false if either field null */

$   select description, unit_price into $d, $up from stock
        where stock_num = $s and manu_code = $m;
    if (sqlca.sqlcode == 0)
        {
            pf_putval(d, CCHARTYPE, de[i]);
            pf_putval(&up, CDECIMALTYPE, pr[i]);
            intreturn(1);
        }
    else
        intreturn(0);
}

valueptr add_order( )
{
$   long custno;
$   long o_date;
$   char ponum[11];
$   char instr[41];
$   long onum;
$   int stno;
$   char manc[4];
$   int quan;
$   dec_t   total;
$   int itno;
$   int i;
$   char errstr[80];

    /* find out whether any items were listed in order */
    pf_getval("tp1", &total, CDECIMALTYPE, 0);
    if (risnull(CDECIMALTYPE, &total))
        {
            pf_msg("No items entered, no order recorded", 0, 1);
            pf_nxfield("sn1");
            lngreturn(0);
        }
}

```



```

        /* collect data for the orders row */
pf_getval("f000", &custno, CLONGTYPE, 0);
pf_getval("f010", &o_date, CLONGTYPE, 0);
pf_getval("f011", ponum, CCHARTYPE, 11);
pf_getval("f012", instr, CCHARTYPE, 41);

        /* begin transaction */
$   begin work;
        /* insert order data into orders table */
$   insert into orders (order_num, customer_num, order_date,
        po_num, ship_instruct)
        values(0, $custno, $o_date, $ponum, $instr);
if (sqlca.sqlcode != 0)
{
    sprintf(errstr, "error number %d in insert", sqlca.sqlcode);
    pf_msg(errstr, 0, 1);
$   rollback work;
    lngreturn(0);
}
onum = sqlca.sqlerrd[1]; /* get serial value assigned */

        /* collect data for items table */
for (i=0; i<5; i++)
{
    pf_getval(sn[i], &stno, CINTTYPE, 0);
    pf_getval(mc[i], manc, CCHARTYPE, 4);
    pf_getval(q[i], &quan, CINTTYPE, 0);
    pf_getval(tp[i], &total, CDECIMALTYPE, 0);
    if (!risnull(CDECIMALTYPE, &total))
    {
        itno = i + 1;
$   insert into items values
        ($itno, $onum, $stno, $manc, $quan, $total);
        if (sqlca.sqlcode != 0)
        {
            sprintf(errstr, "error number %d in update", sqlca.sqlcode);
            pf_msg(errstr, 0, 1);
$   rollback work;
            lngreturn(0);
        }
    }
}
$   commit work;
    lngreturn(onum);
}

```

---

**Figure 6-8. PERFORM Example 2: *st\_desc* and *add\_order* Functions**

# **Appendix A**

## **Header Files**



This appendix contains the following header files:

<b>sqlca.h</b>	is for error codes.
<b>sqllda.h</b>	is for dynamically defined statements.
<b>sqlstype.h</b>	is for SQL statement types.
<b>sqltypes.h</b>	is for SQL and C data types.
<b>decimal.h</b>	is for DECIMAL data types.
<b>ctools.h</b>	is for ACE and PERFORM applications.
<b>datetime.h</b>	is for DATETIME and INTERVAL conversions.

**Note:** If you are using **INFORMIX-OnLine**, additional header files may be required. For information on these header files, see the *INFORMIX-OnLine Programmer's Manual*.



# sqlca.h

---

```
struct sqlca_s
{
    long sqlcode;
    char sqlerrm[72]; /* error message parameters */
    char sqlerrp[8];
    long sqlerrd[6];
        /* 0 - reserved */
        /* 1 - serial value after insert or ISAM error code */
        /* 2 - number of rows processed */
        /* 3 - not used at present */
        /* 4 - offset into SQL of an error */
        /* 5 - rowid after insert */
    struct sqlcaw_s
    {
        char sqlwarn0; /* = W if any of sqlwarn[1-7] = W */
        char sqlwarn1; /* = W if any truncation occurred or
            database has transactions */
        char sqlwarn2; /* = W if a null value returned or
            ANSI database */
        char sqlwarn3; /* = W if no. in select list != no. in into
            list or OnLine backend */
        char sqlwarn4; /* = W if no where clause on prepared update,
            delete or incompatible float format */
        char sqlwarn5; /* = W if non-ANSI statement */
        char sqlwarn6; /* reserved */
        char sqlwarn7; /* reserved */
    } sqlwarn;
};

extern struct sqlca_s sqlca;

#define SQLNOTFOUND 100
```

---

# sqlda.h

---

```
#ifndef _SQLDA
#define _SQLDA

struct sqlvar_struct
{
    short sqltype;        /* variable type */
    short sqllen;         /* length in bytes */
    char *sqldata;        /* pointer to data */
    short *sqlind;        /* pointer to indicator */
    char *sqlname;        /* variable name */
    char *sqlformat;      /* reserved for future use */
    short sqlitype;       /* ind variable type */
    short sqlilen;        /* ind length in bytes */
    char *sqlidata;       /* ind data pointer */
};

struct sqlda
{
    short sqld;
    struct sqlvar_struct *sqlvar;
};

#endif /* _SQLDA */
```

---

# sqlstype.h

---

```
/*
 * SQL statement types
 */

#define SQ_DATABASE      1
#define SQ_SELECT       2
#define SQ_SELINTO      3
#define SQ_UPDATE       4
#define SQ_DELETE       5
#define SQ_INSERT       6
#define SQ_UPDCURR      7
#define SQ_DELCURR      8
#define SQ_LDINSERT     9
#define SQ_LOCK        10
#define SQ_UNLOCK      11
#define SQ_CREADB      12
#define SQ_DROPDB      13
#define SQ_CRETAB      14
#define SQ_DRPTAB      15
#define SQ_CREIDX      16
#define SQ_DRPIDX      17
#define SQ_GRANT        18
#define SQ_REVOKE       19
#define SQ_RENTAB       20
#define SQ_RENCOL       21
#define SQ_CREAUD       22
#define SQ_STRAUD       23
#define SQ_STPAUD       24
#define SQ_DRPAUD       25
#define SQ_RECTAB       26
#define SQ_CHKTAB       27
#define SQ_REPTAB       28
#define SQ_ALTER        29
#define SQ_STATS        30
#define SQ_CLSDB        31
#define SQ_DELALL       32
#define SQ_UPDALL       33
#define SQ_BEGWORK      34
#define SQ_COMMIT       35
#define SQ_ROLLBACK     36
#define SQ_SAVEPOINT    37
```

#define SQ_STARTDB	38
#define SQ_RFORWARD	39
#define SQ_CREVIEW	40
#define SQ_DROPVIEW	41
#define SQ_DEBUG	42
#define SQ_CREASYN	43
#define SQ_DROPSYN	44
#define SQ_CTEMP	45
#define SQ_WAITFOR	46
#define SQ_ALTIDX	47
#define SQ_ISOLATE	48
#define SQ_SETLOG	49
#define SQ_EXPLAIN	50
#define SQ_SCHEMA	51

---



# sqltypes.h

---

```
#ifndef CCHARTYPE
/* C language types */

#define CCHARTYPE          100
#define CSHORTTYPE        101
#define CINTTYPE           102
#define CLONGTYPE          103
#define CFLOATTYPE         104
#define CDOUBLETTYPE       105
#define CDECIMALTYPE       107
#define CFIXCHARTYPE       108
#define CSTRINGTYPE        109
#define CDATETYPE          110
#define CMONEYTYPE         111
#define CDMONEYTYPE        112
#define CLOCATORTYPE       113
#define CVCHARTYPE         114
#define CINVTYPE           115

#define USERCOLL(x)        ((x))

/*
 * Define all possible database types
 * include C-ISAM types here as well as in isam.h
 */

#define SQLCHAR             0
#define SQLSMINT            1
#define SQLINT              2
#define SQLFLOAT            3
#define SQLSMFLOAT          4
#define SQLDECIMAL          5
#define SQLSERIAL           6
#define SQLDATE             7
#define SQLMONEY            8
#define SQLNULL             9
#define SQLDTIME            10
#define SQLBYTES            11
#define SQLTEXT             12
#define SQLVCHAR            13
#define SQLINTERVAL         14
#define SQLTYPE             0xF/* type mask*/
#define SQLNONULL           0400/* disallow nulls*/
```

```

/* this is not a real type but a flag to show that the
 * value is from a host variable
 */
#define SQLHOST 10000
#define SQLNETFLT 02000

#define SIZCHAR 1
#define SIZSMINT 2
#define SIZINT 4
#define SIZFLOAT (sizeof(double))
#define SIZSMFLOAT (sizeof(float))
#define SIZDECIMAL 17/* decimal(32) */
#define SIZSERIAL 4
#define SIZDATE 4
#define SIZMONEY 17/* decimal(32) */
#define SIZDTIME 7/* decimal(12,0) */
#define SIZVCHAR 1

#define ISDECTYPE(t) (((t)&SQLTYPE)==SQLDECIMAL ||
                      ((t)&SQLTYPE)==SQLMONEY ||
                      ((t)&SQLTYPE) == SQLDTIME ||
                      ((t)&SQLTYPE) == SQLINTERVAL)
#define ISBYTESTYPE(type) (((type) & SQLTYPE) == SQLBYTES)
#define ISTEXTTYPE(type) (((type) & SQLTYPE) == SQLTEXT)
#define ISBLOBTYPE(type) (ISBYTESTYPE (type) || ISTEXTTYPE(type))
#define ISBLOBCTYPE(type) ((type) == CLOCATORTYPE)
#define ISVCTYPE(t) (((t) & SQLTYPE) == SQLVCHAR)

#define DEFDECIMAL 9/* default decimal(16) size */
#define DEFMONEY 9/* default decimal(16) size */
#define DATENULL 0x80000000L
#define DATEDOOM 0x80000001L

#define SYSPUBLIC "public"

#endif /* CCHARTYPE */

```

---

# decimal.h

---

```
/*
 * Unpacked Format (format for program usage)
 *
 * Signed exponent "dec_exp" ranging from -64 to +63
 * Separate sign of mantissa "dec_pos"
 * Base 100 digits (range 0 - 99) with decimal point
 * immediately to the left of first digit.
 */

#ifndef DECSIZE
#define DECSIZE      16
#define DECUNKNOWN   -2
#endif

struct decimal
{
    /* exponent base 100 */
    short dec_exp;
    /* sign: 1=pos, 0=neg, -1=null */
    short dec_pos;
    /* number of significant digits */
    short dec_ndgts;
    /* actual digits base 100 */
    char dec_dgts[DECSIZE];
};

typedef struct decimal dec_t;

/*
 * A decimal null is represented
 * internally by setting dec_pos
 * equal to DECPOSNULL
 */
```

```

#define DECPOSNULL      (-1)

/*
 * DECLEN calculates mininum number of bytes
 * necessary to hold a decimal(m,n)
 * where m = total # significant digits and
 * n = significant digits to right of decimal
 */

#define DECLEN      (m,n) (((m)+((n)&1)+3)/2)
#define DECLNGTH(len) DECLEN(PRECTOT(len),PRECDEC(len))

/*
 * DECPREC calculates a default precision given
 * number of bytes used to store number
 */

#define DECPREC(size) (((size-1)<<9)+2)

/* macros to look at and make encoded decimal
 * precision
 *
 * PRECTOT(x)      return total precision
 *                  (digits total)
 * PRECDEC(x)      return decimal precision
 *                  (digits to right)
 * PRECMAKE(x,y)   make precision from total and
 *                  decimal
 */

#define PRECTOT      (x)      (((x)>>8) & 0xff)
#define PRECDEC      (x)      ((x) & 0xff)
#define PRECMAKE(x,y)      (((x)<<8) + (y))

/*
 * Packed Format (format in records in files)
 *
 * First byte =
 *   top 1 bit = sign 0=neg, 1=pos
 *   low 7 bits = Exponent in excess 64 format
 * Rest of bytes = base 100 digits in 100
 *                  complement format
 * Notes -- This format sorts numerically with
 *   just a simple byte by byte unsigned
 *   comparison. Zero is represented as
 *   80,00,00,... (hex). Negative numbers have
 *   the exponent complemented and the base 100
 *   digits in 100's complement
 */

#endif DECSIZE

```

---



# ctools.h

```
/*
 * This is the file which must be included in any C subroutine source
 * file which is to be linked to libsace.a and libsperf.a.
 */

#include "value.h"
#include "datetime.h"
#include "sqltypes.h"

typedef struct value * valueptr;
typedef struct value * acevalue;
typedef struct value * perfvalue;

extern struct value retstack;

#define intreturn(i) {retstack.v_type=SQLSMINT;\
retstack.v_int=(i);\
return(&retstack);}

#define lngreturn(i) {retstack.v_type=SQLINT;\
retstack.v_long=(i);\
return(&retstack);}

#ifndef NOFLOAT
#define floreturn(d) {retstack.v_type=SQLSMFLOAT;\
retstack.v_float=(d);\
return(&retstack);}

#define dubreturn(d) {retstack.v_type=SQLFLOAT;\
retstack.v_double=(d);\
return(&retstack);}

#endif /* NOFLOAT */

#define strreturn(s,c) {retstack.v_type=SQLCHAR;\
retstack.v_charp=(s);\
retstack.v_len=(c);\
return(&retstack);}

#define vcharreturn(s,c) {retstack.v_type=SQLVCHAR;\
retstack.v_charp=(s);\
retstack.v_len=(c);\
return(&retstack);}

#define decreturn(d) {retstack.v_type=SQLDECIMAL;\
deccopy(&d, &retstack.v_decimal);\
return(&retstack);}
```

```

#define dtimereturn(dt)      {retstack.v_type=SQLDTIME;\
                             retstack.v_prec=(dt).dt_qual;\
                             deccopy(&(dt).dt_dec, &retstack.v_decimal);\
                             return(&retstack);}

#define invreturn(inv) {retstack.v_type=SQLINTERVAL;\
                       retstack.v_prec=(inv).in_qual;\
                       deccopy(&(inv).in_dec, &retstack.v_decimal);\
                       return(&retstack);}

extern int toint(); /* toint() takes a pointer to value structure
                   * as an argument.
                   * Returns the value (converted to integer)
                   * of the value structure (v_int).
                   */

extern long tolong(); /* tolong() takes a pointer to value structure
                     * as an argument.
                     * Returns the value (converted to long)
                     * of the value structure (v_long).
                     */

#ifdef NOFLOAT
extern double todouble(); /* todouble() takes a pointer to value structure
                          * as an argument.
                          * Returns the value (converted to double)
                          * of the value structure (v_double).
                          */
#endif /* NOFLOAT */

extern long todate(); /* todate() takes a pointer to value structure
                     * Returns the value (converted to long)
                     * of the value structure (v_long).
                     */

extern int todecimal(); /* todecimal() takes a pointer to a value structure
                        * as the first argument, and a pointer to
                        * a dec_t structure as a second argument.
                        */

extern int todatetime(); /* todatetime(val, dttime, dtqual)
                        * valueptr val;
                        * dttime_t *dttime;
                        * int dtqual;
                        *
                        * input args:val, dtqual
                        * output args:dttime
                        */

extern int tointerval(); /* tointerval(val, intrvl, invqual)
                        * valueptr val;
                        * intrvl_t *intrvl;
                        * int invqual;
                        *
                        * input args:val, invqual
                        * output args:intrvl
                        */

```

```

#define intcon toint
#define longcon      tolong

#ifdef NOFLOAT
#define dubcon todouble
#endif /* NOFLOAT */

struct ufunc
{
    char *uf_id;
    struct value *(*uf_func)();
};

/*
 * The structure declaration for "userfuncs" must be put in the
 * user's data area. A hypothetical case using the user C functions
 * called "userfunc1" and "userfunc2" is shown below.
 *
 * valueptr userfunc1(); These routines must be externed before
 * valueptr userfunc2(); the userfuncs structure is initialized
 *
 * struct ufunc userfuncs[] =
 * {
 *     "userfunc1", userfunc1,
 *     "userfunc2", userfunc2,
 *     ^           ^      Pointer to the user function.
 *     |           |      The name of the user function
 *                       as defined in the DEFINE statement of ACE.
 * 0,0 <-----Note that this array must be terminated
 *                by two zeros.
 * };
 *
 * These structures are required so that ACE can call the user subroutines
 * at run time.
 */

```

---

# datetime.h

---

```
#ifndef DECSIZE
#include "../incl/decimal.h"
#endif

typedef struct dttime
{
    short dt_qual;
    dec_t dt_dec;
} dttime_t;

typedef struct intrvl
{
    short in_qual;
    dec_t in_dec;
} intrvl_t;

/* time units for datetime qualifier */

#define TU_YEAR      0
#define TU_MONTH     2
#define TU_DAY       4
#define TU_HOUR      6
#define TU_MINUTE    8
#define TU_SECOND    10
#define TU_FRAC      12
#define TU_F1        11
#define TU_F2        12
#define TU_F3        13
#define TU_F4        14
#define TU_F5        15

/* TU_END - end time unit in datetime qualifier */
/* TU_START - start time unit in datetime qualifier */
/* TU_LEN - length in digits of datetime qualifier */

#define TU_END(qual) (qual & 0xf)
#define TU_START(qual) ((qual>>4) & 0xf)
#define TU_LEN(qual) ((qual>>8) & 0xff)

/* TU_ENCODE - encode length, start and end time unit to form qualifier */
/* TU_DTENCODE - encode datetime qual */
/* TU_IENCODE - encode interval qual */

#define TU_ENCODE(len,s,e) ((len<<8) | (s<<4) | (e))
#define TU_DTENCODE(s,e) TU_ENCODE((e-s+(s==TU_YEAR?4:2)), s, e) ;
#define TU_IENCODE(len,s,e) TU_ENCODE((e-s+len),s,e);
#define TU_FLEN(len) (TU_LEN(len)-(TU_END(len)-TU_START(len)))

/* TU_CURRQUAL - default qualifier used by current */

#define TU_CURRQUAL TU_ENCODE(17,TU_YEAR,TU_F3)
```

---





# **Appendix B**

## **System Catalogs**

1. *System Catalogs*

2. *System Catalogs*

3. *System Catalogs*

4. *System Catalogs*

# System Catalogs

Information about the database is maintained in the system catalogs. The system catalogs are as follows:

<b>systables</b>	describes database tables.
<b>syscolumns</b>	describes columns in tables.
<b>sysindexes</b>	describes indexes on columns.
<b>sysabauth</b>	identifies table-level privileges.
<b>syscolauth</b>	identifies column-level privileges.
<b>sysdepend</b>	describes how views depend on tables.
<b>sys synonyms</b>	lists synonyms for tables.
<b>sysusers</b>	identifies database-level privileges.
<b>sysviews</b>	defines views.
<b>sysconstraints</b>	records constraints placed on database tables.
<b>sys syntable</b>	used for mapping of synonyms.

The following list gives a brief description of the system catalogs.

## The SYSTABLES Catalog

Column Name	Type	Explanation
tablename	char(18)	name of table
owner	char(8)	owner of table
dirpath	char(64)	directory path for the table file
tabid	serial	internal table identifier
rowsize	smallint	row size
ncols	smallint	number of columns
nindexes	smallint	number of indexes
nrows	integer	number of rows
created	date	date created
version	integer	table version number
tabtype	char(1)	table type (T = table, V = view, S = synonym)
audpath	char(64)	audit filename (full pathname)

Index Name	Type	Columns
tablename	unique	tablename, owner
tabid	unique	tabid



## The SYSCOLUMNS Catalog

Column Name	Type	Explanation
colname	char(18)	column name
tabid	integer	table identifier
colno	smallint	column number
coltype	smallint	column type
collength	smallint	column length (physical)

Index Name	Type	Columns
column	unique	tabid, colno

## The SYSINDEXES Catalog

Column Name	Type	Explanation
idxname	char(18)	index name
owner	char(8)	owner of index
tabid	integer	table identifier
idxtype	char(1)	index type (U = unique, D = dups)
clustered	char(1)	clustering
part1	smallint	column number
part2	smallint	column number
part3	smallint	column number
part4	smallint	column number
part5	smallint	column number
part6	smallint	column number
part7	smallint	column number
part8	smallint	column number

Index Name	Type	Columns
idxtab	dups	tabid
idxname	unique	idxname, tabid

## The SYSTABAUTH Catalog

Column Name	Type	Explanation
grantor	char(8)	grantor of permission
grantee	char(8)	grantee (receiver) of permission
tabid	integer	table identifier
tabauth	char(7)	authorization type

Index Name	Type	Columns
tabgtor	unique	tabid, grantor, grantee
tabgte	dups	tabid, grantee

## The SYSCOLAUTH Catalog

Column Name	Type	Explanation
grantor	char(8)	grantor of permission
grantee	char(8)	grantee (receiver) of permission
tabid	integer	table identifier
colno	smallint	column number
colauth	char(2)	authorization type

Index Name	Type	Columns
colgtor	unique	tabid, grantor, grantee, colno
colgte	dups	tabid, grantee

## The SYSDEPEND Catalog

Column Name	Type	Explanation
btamid	integer	tabid of base table or view
btype	char(1)	base object type (table or view)
dtamid	integer	tabid of dependent table
dtype	char(1)	dependent object type (only view now)

Index Name	Type	Columns
btamid	dups	btamid
dtamid	dups	dtamid

## The SYSSYNONYMS Catalog

Column Name	Type	Explanation
owner	char(8)	user name of owner
synonym	char(18)	synonym identifier
created	date	date synonym created
tabid	integer	table identifier

Index Name	Type	Columns
synonym	unique	owner, synonym
syntabid	dups	tabid

## The SYSUSERS Catalog

Column Name	Type	Explanation
username	char(8)	user login identifier
usertype	char(1)	D = DBA, R = RESOURCE, C = CONNECT
priority	smallint	reserved for future use
password	char(8)	reserved for future use

Index Name	Type	Columns
users	unique	username

## The SYSVIEWS Catalog

Column Name	Type	Explanation
tabid	integer	table identifier
seqno	smallint	sequence number
viewtext	char(64)	portion of SELECT statement

Index Name	Type	Columns
view	unique	tabid, seqno



## The SYSCONSTRAINTS Catalog

Column Name	Type	Explanation
constrid	serial	reserved for future use
constrname	char(18)	constraint identifier
owner	char(8)	user name of owner
tabid	integer	table identifier
constrtype	char(1)	reserved for future use
idxname	char(18)	index name

Index Name	Type	Columns
constrname	unique	constrname, owner
constrtabid	dups	tabid
constrid	unique	constrid

## The SYSSYNTABLE Catalog

Column Name	Type	Explanation
tabid	integer	table identifier
servername		used on INFORMIX-OnLine only
dbname		used on INFORMIX-OnLine only
tabname		used on INFORMIX-OnLine only
owner		used on INFORMIX-OnLine only
btbid	integer	tabid of base table or view

Index Name	Type	Columns
synntabid	unique	tabid
synnbtbid	dups	btbid

# **Appendix C**

## **Environment Variables**



# Environment Variables

**INFORMIX-ESQL/C** makes the following assumptions about the user's environment:

- The editor used is the predominant editor for the operating system (usually **vi**).
- The database worked with is in the current directory.
- The program that sends files to the printer is **lp**.
- The directory used to store temporary files is **/tmp**.
- The **INFORMIX-ESQL/C** compiler and its associated files and libraries are located in **/usr/informix**.



# Setting Environment Variables

You can change any of the assumptions by setting one or more of the environment variables recognized by INFORMIX-ESQL/C. You can set environment variables at the system prompt or in your **.login** (C shell) or **.profile** (Bourne shell) file.

- When you set an environment variable at the system prompt, you must reassign it the next time you log onto the system.
- When you set a variable in your **.login** (C shell) or **.profile** (Bourne shell) file, it is assigned automatically every time you log onto the system.

Enter commands to set the **ABCD** environment variable to *value* as follows:

## C shell:

```
setenv ABCD value
```

## Bourne shell:

```
ABCD=value  
export ABCD
```

The environment variables recognized by INFORMIX-ESQL/C are as follows:

**DBANSIWARN** specifies that you want to initiate Informix extension checking. Setting the **DBANSIWARN** environment variable before you compile an **INFORMIX-ESQL/C** program is functionally equivalent to compiling with the **-ansi** flag. When Informix extensions to ANSI standard syntax are encountered in your program, warning messages are written to the screen. At run time, the **DBANSIWARN** environment variable causes **sqlwarn5** to be set to **W** when a non-ANSI statement is executed.

Unlike most environment variables, you do not set **DBANSIWARN** to a value. Simply setting it in the environment is sufficient. Set the **DBANSIWARN** environment variable as follows:

### **C shell:**

```
setenv DBANSIWARN
```

### **Bourne shell:**

```
DBANSIWARN=  
export DBANSIWARN
```

Once you have set DBANSIWARN, Informix extension checking is automatic until you log out or unset DBANSIWARN.

When you want to turn off Informix extension checking, you can unset the DBANSIWARN environment variable using the following command:

### **C shell:**

```
unsetenv DBANSIWARN
```

### **Bourne shell:**

```
unset DBANSIWARN
```

## **DBDATE**

specifies the format you want to use for DATE values. Through DBDATE, you can specify

- The order of the month, day, and year in a date
- Whether the year should be printed with two digits (Y2) or four digits (Y4)
- The separator between the month, day, and year

The default value for DBDATE is

```
MDY4/
```

where M represents the month, D represents the day, Y4 represents a four-digit year, and the separator is a ( / ) slash (mm/dd/yyyy).

Other acceptable values for the separator are a hyphen ( - ), a period ( . ), or a zero (0). The zero causes the date to be displayed without a separator. If you attempt to use any values other than the hyphen, period, or zero, the separator is a slash. If you do not specify a separator, the date is displayed with a slash ( / ) separator.

Suppose you want the date to appear in European format (dd-mm-yyyy). Set the DBDATE environment variable as follows:

**C shell:**

```
setenv DBDATE DMY4-
```

**Bourne shell:**

```
DBDATE=DMY4-  
export DBDATE
```

**DBMONEY**

applies to MONEY values. It has the format

*[front]*[. |,]*[back]*

where *front* is the optional symbol that precedes the MONEY value, the comma or the period is the optional symbol that separates the integral from the fractional part of the MONEY value, and *back* is the optional symbol that follows the MONEY value. The *front* and *back* symbols can be up to seven characters long and can contain any characters except commas or periods.

The default value for DBMONEY is

\$.

where the dollar sign precedes the MONEY value, and the period ( . ) separates the integral from the fractional part of the MONEY value. Suppose you want to represent MONEY values in *deutsche marks*.

Set the DBMONEY environment variable as follows:

**C shell:**

```
setenv DBMONEY DM,
```

**Bourne shell:**

```
DBMONEY=DM,  
export DBMONEY
```

where DM is the currency symbol, and the comma separates the integral from the fractional part of the MONEY value.

DBPATH	specifies a list of directories for INFORMIX-ESQL/C to search for databases and associated files. When looking for a database and related files, INFORMIX-ESQL/C first checks the current directory and then checks the directories specified in DBPATH. Use the same format that you use to set the PATH variable.
DBPRINT	specifies the print program for your computer. For most systems, the default program is lp.
DBTEMP	specifies the directory into which INFORMIX-ESQL/C places its temporary files. You need not set DBTEMP if the default, /tmp, is satisfactory.
INFORMIXDIR	specifies the directory containing the INFORMIX-ESQL/C files.
SQLEXEC	directs your front-end processes to the appropriate database engine. By default, these processes look first for the INFORMIX-OnLine database engine. Therefore, you must set SQLEXEC only if you have both the INFORMIX-SE and INFORMIX-OnLine database engines installed on your system, and you want to access INFORMIX-SE.



SQLEXEC must contain the full pathname of the database engine, which is found in the **/lib** subdirectory of your \$INFORMIXDIR directory.

To specify the **INFORMIX-OnLine** engine:

**C shell:**

```
setenv SQLEXEC $INFORMIXDIR/lib/sqlturbo
```

**Bourne shell:**

```
SQLEXEC=$INFORMIXDIR/lib/sqlturbo  
export SQLEXEC
```

To specify the **INFORMIX-SE** engine:

**C shell:**

```
setenv SQLEXEC $INFORMIXDIR/lib/sqlexec
```

**Bourne shell:**

```
SQLEXEC=$INFORMIXDIR/lib/sqlexec  
export SQLEXEC
```

# **Appendix D**

## **Reserved Words**



# Reserved Words

This appendix contains a list of Informix reserved words. Do not use these words as identifiers or as program variable or host variable names. If you use reserved words as identifiers or variables, your program (or SQL statement) may fail with an error.

This list contains reserved words from all Informix products, although not all are reserved in each product. Note that, while their use is not recommended, some reserved words may not cause errors in every case. For example, words that are reserved in INFORMIX-4GL will not generate an error if used with only INFORMIX-SQL. However, if your INFORMIX-SQL application is later ported to an INFORMIX-4GL environment, any INFORMIX-4GL reserved words will cause errors. Additionally, some words generate errors only if used as host or program variables, while other words generate errors only if used as identifiers.

In addition to the words on this list, you should not use C, Ada, COBOL, or FORTRAN language keywords in your programs.

abort	avg	close
absolute	background	cluster
accept	before	col
access	begin	color
add	beginning	colors
after	bell	column
all	between	columns
allowing	black	command
alter	blanks	comment
and	blink	comments
ansi	blue	commit
any	bold	committed
array	border	composites
as	bottom	compress
asc	buffered	connect
ascending	by	constant
ascii	byte	constraint
at	call	construct
attribute	case	continue
attributes	char	count
audit	character	convert
auto	check	create
autonext	clear	current
average	clipped	cursor



cyan  
database  
date  
datetime  
date\_type  
day  
dba  
debug  
dec  
decimal  
decimal\_type  
declare  
dec\_t  
default  
defaults  
defer  
define  
delete  
delimiter  
delimiters  
desc  
descending  
describe  
descriptor  
dim  
dirty  
display  
displayonly  
distinct  
do  
dominant  
double  
down  
downshift  
drop  
dtype  
dtype\_t  
eco-  
editadd  
editupdate  
else  
end  
end-exec  
endif  
ending

error  
escape  
every  
exclusive  
exec  
execute  
exists  
exit  
exitnow  
exits  
explain  
extend  
extent  
extern  
external  
false  
fetch  
field  
file  
finish  
first  
fixchar  
float  
flush  
for  
foreach  
form  
format  
formonly  
found  
fraction  
free  
from  
function  
globals  
go  
go to  
goto  
grant  
green  
group  
having  
header  
headings  
help

hold  
hour  
identified  
if  
ifdef  
ifndef  
immediate  
in  
include  
index  
indicator  
infield  
info  
initialize  
input  
insert  
instructions  
int  
integer  
interrupt  
intersect  
interval  
into  
intrvl\_t  
inverse  
invisible  
is  
isam  
isolation  
join  
joining  
key  
label  
last  
left  
len  
length  
let  
like  
line  
lineno  
lines  
load  
locator  
lock

loc\_t  
log  
long  
long\_float  
long\_integer  
lookup  
loop  
magenta  
main  
margin  
master  
matches  
max  
mdy  
memory  
menu  
message  
min  
minus  
minute  
mod  
mode  
modify  
module  
money  
month  
name  
natural  
need  
new  
next  
nextfield  
no  
nocr  
noentry  
normal  
not  
not found  
notfound  
nouupdate  
now  
null  
numeric  
of  
off

on  
open  
option  
options  
or  
order  
otherwise  
out  
outer  
output  
package  
page  
pageno  
param  
pause  
percent  
perform  
picture  
pipe  
positive  
precision  
prepare  
previous  
print  
printer  
prior  
privilege  
privileges  
program  
prompt  
public  
put  
query  
queryclear  
quit  
raise  
range  
read  
readonly  
real  
record  
recover  
red  
register  
relative

remove  
rename  
repair  
repeatable  
report  
required  
resource  
return  
returning  
reverse  
revoke  
right  
rollback  
rollforward  
row  
rowid  
rows  
run  
savepoint  
screen  
scroll  
second  
section  
select  
serial  
serial\_type  
set  
share  
shift  
short  
short\_float  
short\_integer  
sitename  
size  
skip  
sleep  
smallfloat  
smallint  
some  
space  
spaces  
sql  
sql\*  
sqlca  
sqlchar\_type

sqllda  
sqldecimal\_type  
sqlerr  
sqlerror  
sqlfloat\_type  
sqlint\_type  
sqlmoney\_type  
sqlsmfloat\_type  
sqlsmint\_type  
sqlwarning  
stability  
start  
startlog  
static  
statistics  
status  
stdv  
step  
stop  
string  
struct  
subtract  
subtype  
sum  
synonym  
systables  
table

temp  
text  
then  
through  
thru  
time  
tiny\_integer  
to  
today  
top  
total  
trailer  
trailing  
true  
type  
typedef  
undef  
underline  
union  
unique  
units  
unload  
unlock  
up  
update  
upshift  
user

using  
validate  
values  
varchar  
variable  
vc\_t  
verify  
view  
wait  
waiting  
warning  
weekday  
when  
whenever  
where  
while  
white  
window  
with  
without  
wordwrap  
work  
wrap  
year  
yellow  
yes  
zerofill

# **Appendix E**

## **The *stores* Database**





# The *stores* Database

The **stores** database contains a set of tables that describe an imaginary business. You can access the data in **stores** by the demonstration programs that appear in this book, as well as by application programs that are listed in the documentation of other Informix products. The **stores** database is not MODE ANSI.

This appendix contains four sections:

- The first section describes the structure of the tables in the **stores** database. For each table, the name and the data type of each column are listed. Any indexes on individual columns or on multiple columns are identified and classified as unique or as allowing duplicate values.
- The second section presents a graphic map of the tables in the **stores** database, showing potential join columns.
- The third section discusses the join columns that link some of the tables in the **stores** database, and illustrates how you can use these relationships to obtain information from multiple tables.
- The final section shows the data contained in each table of the **stores** database.

If you have modified or deleted some or all of the data in these tables, you can restore the **stores** database to its original form by running the **esqldemo** script.

## Structure of the Tables

The **stores** database contains information about a fictitious sporting goods distributor that services stores in the Western United States. This database includes the following tables:

- **customer**
- **orders**
- **items**
- **stock**
- **manufact**
- **state**

### The *customer* Table

The **customer** table describes retail stores that order from the distributor. Information includes each store's name, address, and telephone number, and the name of its representative. The columns of the **customer** table are as follows:

customer_num	serial(101)
fname	char(15)
lname	char(15)
company	char(20)
address1	char(20)
address2	char(20)
city	char(15)
state	char(2)
zipcode	char(5)
phone	char(18)

The **customer\_num** column is indexed as unique.

The **zipcode** column is indexed to allow duplicate values.

### The *orders* Table

The **orders** table contains information about orders placed by the distributor's customers. This information includes the order number, the date the order was made, the customer number, the shipping instructions, the customer's purchase order number, the date the order was shipped, the weight of the order, the shipment charge, and the date the customer paid for the order. It also tells whether a backlog exists. The columns of the **orders** table are as follows:

order_num	serial(1001)
order_date	date
customer_num	integer
ship_instruct	char(40)
backlog	char(1)
po_num	char(10)
ship_date	date
ship_weight	decimal(8,2)
ship_charge	money(6)
paid_date	date

The **order\_num** column is indexed and must contain unique values. The **customer\_num** column is indexed but allows duplicates.

## The *items* Table

An order can include one or more items. The **items** table describes the items in each order. Information in the **items** table includes the item number, codes for the order, stock, and manufacturer, the quantity, and the total price for the item. The columns of the **items** table are as follows:

item_num	smallint
order_num	integer
stock_num	smallint
manu_code	char(3)
quantity	smallint
total_price	money(8)

The **order\_num** column has an index that allows duplicate values. A multiple-column index for the **stock\_num** and **manu\_code** columns also permits duplicate values.

## The *stock* Table

The distributor carries 15 different types of sporting goods. For example, the distributor offers baseball gloves from three manufacturers, and basketballs from one manufacturer.

The **stock** table is a catalog of the items sold by the distributor. For each item in stock, the **stock** table lists a stock number identifying the type of item, a code identifying the manufacturer, a description of the item, its unit price, the unit by which the item must be ordered, and a description of the unit (for example, a case containing 10 baseballs).



The names and data types of the columns in the **stock** table are as follows:

stock_num	smallint
manu_code	char(3)
description	char(15)
unit_price	money(6)
unit	char(4)
unit_descr	char(15)

A multiple-column index for both the **stock\_num** and **manu\_code** columns allows only unique values.

## The *manufact* Table

Information about the five manufacturers whose sporting goods are handled by the distributor is kept in the **manufact** table. Each row contains a manufacturer's identifying code and name. The columns in the **manufact** table are as follows:

manu_code	char(3)
manu_name	char(15)

The **manu\_code** column has an index that requires unique values.

## The *state* Table

The **state** table contains the names and postal abbreviations of the fifty states of the United States. It includes two columns:

code	char(2)
sname	char(15)

The **code** column is indexed as unique.

## Map of the stores Database

Figure E-1 displays the column names of the tables in the stores database, and indicates which columns in different tables contain the same information.

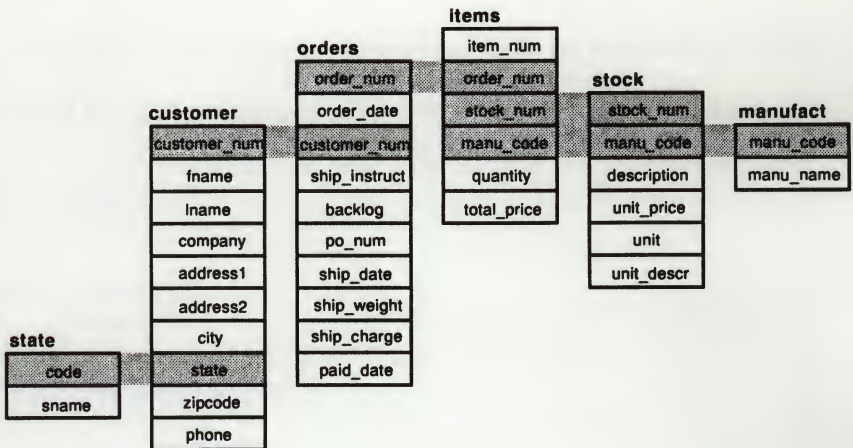


Figure E-1. Tables in the stores Database

## Join Columns That Link the Database

The tables of the **stores** database are linked together by *join columns*, which are identified in this section. You can use them to retrieve and display information from several tables at once, as if it had been stored in a single table. Figures E-2 through E-6 show the relationships among tables, and how information stored in one table supplements information in others.

### Join Columns in the *customer* and *orders* Tables

The **customer\_num** column joins the **customer** table and the **orders** table, as shown in Figure E-2.

---

			customer table (detail)
customer_num	fname	lname	
101	Ludwig	Pauli	
102	Carole	Sadler	
103	Philip	Currie	
104	Anthony	Higgins	

			orders table (detail)
order_num	order_date	customer_num	
1001	01/20/1989	104	
1002	06/01/1989	101	
1003	10/12/1989	104	
1004	04/12/1989	106	

---

**Figure E-2.** Tables Joined by the **customer\_num** Column

The **customer** table contains a **customer\_num** column that holds a number identifying a customer, along with columns for the customer's name, company, address, and telephone number. For example, the row with information about Anthony Higgins contains the number 104 in the **customer\_num** column.

The **orders** table also contains a **customer\_num** column that stores the number of the customer who placed a particular order. According to Figure E-2, customer 104 (Anthony Higgins) has placed two orders, since his customer number appears in two rows of the **orders** table.

The join relationship lets you select information from both tables. This means that you can retrieve Anthony Higgins's name and address and information about his orders at the same time.



# Join Columns in the *orders* and *items* Tables

The **orders** and **items** tables are linked by an **order\_num** column that contains an identification number for each order. If an order includes several items, the same order number appears in several rows of the **items** table. Figure E-3 shows this relationship.

**orders table**  
(detail)

order_num	order_date	customer_num
1001	01/20/1989	104
1002	06/01/1989	101
1003	10/12/1989	104

**items table**  
(detail)

item_num	order_num	stock_num	manu_code
1	1001	1	HRO
1	1002	4	HSK
2	1002	3	HSK
1	1003	9	ANZ
2	1003	8	ANZ
3	1003	5	ANZ

**Figure E-3.** Tables Joined by the **order\_num** Column

## Join Columns in the *items* and *stock* Tables

The *items* table and the *stock* table are joined by two columns: the **stock\_num** column stores a stock number for an item, and the **manu\_code** column stores a code to identify the manufacturer. You need both the stock number and the manufacturer code to uniquely identify an item. For example, the item with the stock number 1 and the manufacturer code HRO is a Hero baseball glove, while the item with the stock number 1 and the manufacturer code HSK is a Husky baseball glove.

items table (detail)			
item_num	order_num	stock_num	manu_code
1	1001	1	HRO
1	1002	4	HSK
2	1002	3	HSK
1	1003	9	ANZ
2	1003	8	ANZ
3	1003	5	ANZ
1	1004	1	HRO

stock table (detail)		
stock_num	manu_code	description
1	HRO	baseball glove
1	HSK	baseball glove
1	SMT	baseball glove

**Figure E-4.** Tables Joined by Two Columns

The same stock number and manufacturer code can appear in more than one row of the *items* table, if the same item belongs to separate orders. This is illustrated in Figure E-4.

## Join Columns in the *stock* and *manufact* Tables

The **stock** table and the **manufact** table are joined by the **manu\_code** column. The same manufacturer code can appear in more than one row of the **stock** table if the manufacturer produces more than one piece of equipment. This relationship is illustrated in Figure E-5.

stock_num	manu_code	description	stock table (detail)
1	HRO	baseball glove	
1	HSK	baseball glove	
1	SMT	baseball glove	
2	HRO	baseball	

manu_code	manu_name	manufact table (detail)
NRG	Norge	
HSK	Husky	
HRO	Hero	

**Figure E-5.** Tables Joined by the **manu\_code** Column



## Join Columns in the *state* and *customer* Tables

The **state** table and the **customer** table are joined by a column that contains the state code. This column is called **code** in the **state** table and **state** in the **customer** table. If several customers live in the same state, the same state code will appear in several rows of the **customer** table, as shown in Figure E-6.

					customer table (detail)	
customer_num	fname	lname	...	state		
101	Ludwig	Pauli	...	CA		
102	Carole	Sadler	...	CA		
103	Philip	Currie	...	CA		

code	sname			state table (detail)	
AK	Alaska				
AL	Alabama				
AR	Arkansas				
AZ	Arizona				
CA	California				

**Figure E-6.** Tables Joined by the State-Code Column

Joining lets you rearrange your view of a database whenever you want. It provides flexibility that lets you create new relationships between tables without redesigning the database. You can easily expand the scope of a database by adding new tables that join the existing tables. As you read through this manual, you will see programs that set up the join relationships across tables of the **stores** database. Refer to these figures whenever you need to review these relationships.



## ***Data in the stores Database***

The tables that follow display the data in the **stores** database.

customer Table

customer_num	fname	lname	company	address1	address2	city	state	zipcode	phone
101	Ludwig	Pauli	All Sports Supplies	213 Erstwild Court		Sunnyvale	CA	94086	408-789-8075
102	Carole	Sadler	Sports Spot	785 Geary St		San Francisco	CA	94117	415-822-1289
103	Philip	Currie	Phil's Sports	654 Poplar	P. O. Box 3498	Palo Alto	CA	94303	415-328-4543
104	Anthony	Higgins	Play Ball!	East Shopping Cntr.	422 Bay Road	Redwood City	CA	94026	415-368-1100
105	Raymond	Vector	Los Altos Sports	1899 La Loma Drive		Los Altos	CA	94022	415-776-3249
106	George	Watson	Watson & Son	1143 Carver Place		Mountain View	CA	94063	415-389-8789
107	Charles	Ream	Athletic Supplies	41 Jordan Avenue		Palo Alto	CA	94304	415-356-9876
108	Donald	Quinn	Quinn's Sports	587 Alvarado		Redwood City	CA	94063	415-544-8729
109	Jane	Miller	Sport Stuff	Mayfair Mart	7345 Ross Blvd.	Sunnyvale	CA	94086	408-723-8789
110	Roy	Jaeger	AA Athletics	520 Topaz Way		Redwood City	CA	94062	415-743-3611
111	Frances	Keyes	Sports Center	3199 Sterling Court		Sunnyvale	CA	94085	408-277-7245
112	Margaret	Lawson	Runners & Others	234 Wyandotte Way		Los Altos	CA	94022	415-887-7235
113	Lana	Beatty	Sportstown	654 Oak Grove		Menlo Park	CA	94025	415-356-9982
114	Frank	Albertson	Sporting Place	947 Waverly Place		Redwood City	CA	94062	415-886-6677
115	Alfred	Grant	Gold Medal Sports	776 Gary Avenue		Menlo Park	CA	94025	415-356-1123
116	Jean	Parnelee	Olympic City	1104 Spinosa Drive		Mountain View	CA	94040	415-534-8822
117	Arnold	Sipes	Kids Korner	850 Lytton Court		Redwood City	CA	94063	415-245-4578
118	Dick	Baxter	Blue Ribbon Sports	5427 College		Oakland	CA	94609	415-655-0011

orders Table

order_num	order_date	customer_num	ship_instruct	backlog	po_num	ship_date	ship_weight	ship_charge	paid_date
1001	01/20/89	104	ups	n	B77836	02/01/89	20.40	10.00	03/22/89
1002	06/01/89	101	po on box; deliver back door only	n	9270	06/06/89	50.60	15.30	07/03/89
1003	10/12/89	104	via ups	n	B77890	10/13/89	35.60	10.80	11/04/89
1004	04/12/89	106	ring bell twice	y	8006	04/30/89	95.80	19.20	
1005	12/04/89	116	call before delivering	n	2865	12/19/89	80.80	16.20	12/30/89
1006	09/19/89	112	after 10 am	y	Q13557		70.80	14.20	
1007	03/25/89	117		n	278693	04/23/89	125.90	25.20	
1008	11/17/89	110	closed Monday	y	L2230	12/06/89	45.60	13.80	12/21/89
1009	02/14/89	111	door next to supersaver	n	4745	03/04/89	20.40	10.00	04/21/89
1010	05/29/89	115	deliver 776 Gary if no answer	n	429Q	06/08/89	40.60	12.30	07/22/89
1011	03/23/89	104	ups	n	B77897	04/13/89	10.40	5.00	06/01/89
1012	06/05/89	117		n	278701	06/09/89	70.80	14.20	
1013	09/01/89	104	via ups	n	B77930	09/18/89	60.80	12.20	10/10/89
1014	05/01/89	106	ring bell, kick door loudly	n	8052	05/10/89	40.60	12.30	07/18/89
1015	07/10/89	110	closed Mon	n	MA003	08/01/89	20.60	6.30	08/31/89

items Table

item_num	order_num	stock_num	manu_code	quantity	total_price
1	1001	1	HRO	1	250.0
1	1002	4	HSK	1	960.0
2	1002	3	HSK	1	240.0
1	1003	9	ANZ	1	20.0
2	1003	8	ANZ	1	840.0
3	1003	5	ANZ	5	99.0
1	1004	1	HRO	1	960.0
2	1004	2	HRO	1	126.0
3	1004	3	HSK	1	240.0
4	1004	1	HSK	1	800.0
1	1005	5	NRG	10	280.0
2	1005	5	ANZ	10	198.0
3	1005	6	SMT	1	36.0
4	1005	6	ANZ	1	48.0
1	1006	5	SMT	5	125.0
2	1006	5	NRG	5	190.0
3	1006	5	ANZ	5	99.0
4	1006	6	SMT	1	36.0
5	1006	6	ANZ	1	48.0
1	1007	1	HRO	1	250.0
2	1007	2	HRO	1	126.0
3	1007	3	HSK	1	240.0
4	1007	4	HRO	1	480.0
5	1007	7	HRO	1	600.0
1	1008	8	ANZ	1	840.0
2	1008	9	ANZ	5	100.0
1	1009	1	SMT	1	450.0
1	1010	6	SMT	1	36.0
2	1010	6	ANZ	1	48.0
1	1011	5	ANZ	5	99.0
1	1012	8	ANZ	1	840.0
2	1012	9	ANZ	10	200.0
1	1013	5	ANZ	1	19.8
2	1013	6	SMT	1	36.0
3	1013	6	ANZ	1	48.0
4	1013	9	ANZ	2	40.0
1	1014	4	HSK	1	960.0
2	1014	4	HRO	1	480.0
1	1015	1	SMT	1	450.0



**stock Table**

stock_num	manu_code	description	unit_price	unit	unit_descr
1	HRO	baseball gloves	250.00	case	10 gloves/case
1	HSK	baseball gloves	800.00	case	10 gloves/case
1	SMT	baseball gloves	450.00	case	10 gloves/case
2	HRO	baseball	126.00	case	24/case
3	HSK	baseball bat	240.00	case	12/case
4	HSK	football	960.00	case	24/case
4	HRO	football	480.00	case	24/case
5	NRG	tennis racquet	28.00	each	each
5	SMT	tennis racquet	25.00	each	each
5	ANZ	tennis racquet	19.80	each	each
6	SMT	tennis ball	36.00	case	24 cans/case
6	ANZ	tennis ball	48.00	case	24 cans/case
7	HRO	basketball	600.00	case	24/case
8	ANZ	volleyball	840.00	case	24/case
9	ANZ	volleyball net	20.00	each	each

**manufact Table**

manu_code	manu_name
ANZ	Anza
HSK	Husky
HRO	Hero
NRG	Norge
SMT	Smith

# state Table

code	sname	code	sname
AK	Alaska	MT	Montana
AL	Alabama	NB	Nebraska
AR	Arkansas	NC	North Carolina
AZ	Arizona	ND	North Dakota
CA	California	NH	New Hampshire
CT	Connecticut	NJ	New Jersey
CO	Colorado	NM	New Mexico
DE	Delaware	NV	Nevada
FL	Florida	NY	New York
GA	Georgia	OH	Ohio
HI	Hawaii	OK	Oklahoma
IA	Iowa	OR	Oregon
ID	Idaho	PA	Pennsylvania
IL	Illinois	RI	Rhode Island
IN	Indiana	SC	South Carolina
KS	Kansas	SD	South Dakota
KY	Kentucky	TN	Tennessee
LA	Louisiana	TX	Texas
MA	Massachusetts	UT	Utah
MD	Maryland	VA	Virginia
ME	Maine	VT	Vermont
MI	Michigan	WA	Washington
MN	Minnesota	WI	Wisconsin
MO	Missouri	WV	West Virginia
MS	Mississippi	WY	Wyoming



# **Appendix F**

## **INFORMIX-ESQL/C Utility Programs**





## What This Appendix Contains

This appendix describes the seven utility programs included with the **INFORMIX-ESQL/C** software:

- The **bcheck** utility checks and restores the integrity of your index files.
- The **dbload** utility allows you to load data from other database systems or from raw data files into **INFORMIX-ESQL/C** databases.
- The **dbschema** utility allows you to output the SQL statements necessary to replicate an individual table or an entire database.
- The **dbupdate** utility lets you convert an older version of the SQL database to the current structure.
- The **sqlconv** utility converts an **INFORMIX** database to an SQL-compatible database.
- The **dbexport** utility allows you to unload a database into ASCII files for import into another database environment.
- The **dbimport** utility allows you to create a database from ASCII files.

---

**INFORMIX-OnLine** supports additional functionality. Refer to the *INFORMIX-OnLine Programmer's Manual* for more information.

---

# The *bcheck* Utility

The **bcheck** program is a C-ISAM utility program that checks and repairs C-ISAM index files. It is distributed with **INFORMIX-ESQL/C** on **INFORMIX-SE**.

**bcheck** compares an index (.idx) file to a data (.dat) file to see if the two are consistent. If they are not, **bcheck** asks whether you want to delete and rebuild the corrupted indexes.

When running **bcheck** from the operating system command line, you must specify the table name used by the **INFORMIX-ESQL/C** system catalogs. For example, the **customer** table in the **stores** database is identified in the system catalogs as **custome100** and not as **customer**.

You can list the contents of the database directory to determine the appropriate table name.

In the following example, **bcheck** is run from the command line on the **customer** file and finds no errors.

---

```
bcheck -n custome100
```

```
BCHECK C-ISAM B-tree Checker version 4.00.00
Copyright (C) 1981-1989 Informix Software, Inc.
Software Serial Number INF#R000000
```

```
C-ISAM File: custome100
```

```
Checking dictionary and file sizes.
Index file node size = 1024
Current C-ISAM index file node size = 1024
Checking data file records.
Checking indexes and key descriptions.
Index 1 = unique key
    0 index node(s) used -- 1 index b-tree level(s) used
Index 2 = unique key (0,4,2)
    1 index node(s) used -- 1 index b-tree level(s) used
Index 3 = duplicates (111,5,0)
    1 index node(s) used -- 1 index b-tree level(s) used
Checking data record and index node free lists.
4 index node(s) used, 0 free -- 18 data record(s) used, 4 free
```

---

For each index, **bcheck** prints a group of up to eight numbers. These numbers indicate the position of the key in each record.

You can also use **bcheck** with the following options:

- i Check index file only
- l List entries in B+ trees
- n Answer no to all questions
- y Answer yes to all questions
- q Suppress printing of the program banner
- s Resize the index file node size

The **bcheck** command syntax is as follows:

---

```
bcheck- [i | l | y | n | q | s] file-name
```

---

Unless you use the **-n** or **-y** option, **bcheck** is interactive, waiting for you to respond to each error it finds.

Use the **-y** option with caution. Do not run **bcheck** using the **-y** option if you are checking the files for the first time.



Here is a sample run in which **bcheck** finds errors. The **-n** option is selected, so that each question **bcheck** asks is automatically answered "no."

---

BCHECK C-ISAM B-tree Checker version 4.00.00  
Copyright (C) 1981-1989 Informix Software, Inc.  
Software Serial Number INF#R000000

C-ISAM File: custome100

Checking dictionary and file sizes.

Index file node size = 1024

Current C-ISAM index file node size = 1024

Checking data file records.

Checking indexes and key descriptions.

Index 1 = unique key

0 index node(s) used -- 1 index b-tree level(s) used

ERROR: 3 bad data record(s)

Delete index ? no

Index 2 = unique key (0,4,2)

1 index node(s) used -- 1 index b-tree level(s) used

ERROR: 3 bad data record(s)

Delete index ? no

Index 3 = duplicates (111,5,0)

1 index node(s) used -- 1 index b-tree level(s) used

ERROR: 3 bad data record(s)

Delete index ? no

Checking data record and index node free lists.

ERROR: 3 missing data record(s)

Fix data record free list ? no

4 index node(s) used, 0 free -- 18 data record(s) used, 4 free

---

Since **bcheck** finds errors, you must delete and rebuild the corrupted indexes. The **-y** option is used to answer "yes" to all questions asked by **bcheck**:

---

BCHECK C-ISAM B-tree Checker version 4.00.00  
Copyright (C) 1981-1989 Informix Software, Inc.  
Software Serial Number INF#R000000

C-ISAM File: custome100

Checking dictionary and file sizes.

Checking data file records.

Checking indexes and key descriptions.

Index 1 = unique key

1 index node(s) used -- 1 index b-tree level(s) used

ERROR: 3 bad data record(s)

Delete index ? yes

Remake index ? yes

Index 2 = unique key (0,4,2)

1 index node(s) used -- 1 index b-tree level(s) used

ERROR: 3 bad data record(s)

Delete index ? yes

Remake index ? yes

Index 3 = duplicates (111,5,0)

1 index node(s) used -- 1 index b-tree level(s) used

ERROR: 3 bad data record(s)

Delete index ? yes

Remake index ? yes

Checking data record and index node free lists.

ERROR: 3 missing data record(s)

Fix data record free list ? yes

Recreate data record free list

Recreate index 3

Recreate index 2

Recreate index 1

4 index node(s) used, 0 free -- 18 data record(s) used, 4 free

---

# The *dbload* Utility

The **dbload** utility provides a method for transferring data from ASCII files into an existing database. This program supports the easy and efficient transfer of database files created for other Informix products, or even for entirely different database management systems. The **dbload** utility includes the following features:

- Data from selected fields of one or more input files can be loaded into selected columns of one or more database tables.
- Loading can begin at any line in the input file.
- Loading proceeds in batches of  $n$  records (where  $n$  is an integer that you specify).
- Both fixed- and variable-length data records can be loaded.
- NULL values can be defined for any field of a record.
- Constants that are not in the data records can be loaded.
- Records that cannot be loaded into the database are trapped and stored (with diagnostic information) in an error log file.
- The user can specify an error limit, and **dbload** stops when that limit is reached on the number of records that cannot be entered into the database because of errors.
- If your database supports transactions, you have the option of terminating the loading process without committing any data from the batch of records that exceeded your error limit.

To use **dbload**, you must have at least one ASCII *input file* of data records to enter, and at least one *table* to receive the data. You must then create a *command file* to specify instructions for reading and loading the data. Finally, you must invoke **dbload** by entering an appropriate *command line*. The following sections provide details about these procedures.



## *Input Files for the dbload Utility*

Just as database tables store data in columns within rows, input file data must be arranged in *fields* within *records*. You must be able to specify a one-to-one correspondence between fields of the input records and columns of the new rows that **dbload** will insert in the database.

Data files for **dbload** must be “flat” ASCII files, containing only printable characters. The records containing the data to be transferred must be separated by the **NEWLINE** character. Output from the **UNLOAD** statement of SQL, for example, can be used as an input file by **dbload**.

### **Fixed-Length and Variable-Length Records**

The record length can be either fixed or variable. An input file has *fixed-length* records if the locations of data fields and the number of characters are the same across all records. If a file has *variable-length* records, every field must end with the same delimiter character (which must not occur as data within any field). This character does not need to be the same one that the **DBDELIMITER** environment variable specifies.

Two consecutive delimiters in a variable-length record define a **NULL** field. For each field in a fixed-length record, you can use the **dbload** command file to specify a character string to store as a **NULL** value.

### **Data Type Formats**

Leading blanks are allowed in data fields. A currency symbol is optional in data fields for **MONEY** columns.

Values of type **DATE** must be in *mm / dd / yyyy* format. Data for **DATETIME** and **INTERVAL** columns must be in character form, showing only field digits and delimiters (no type or qualifiers):

*yyyy - mm - dd hh : mi : ss . fff*

Here *yyyy* represents *year* digits, *mm* the *month* (January = 1 or 01), *dd* the *day* of the month, *hh* the *hour*, *mi* the *minute*, *ss* the *second*, and *fff* the *fractional* part of a second.



## Specifying a dbload Command File

Before you can use **dbload**, you must first create an ASCII *command file* that maps fields from one or more input files into columns of one or more tables within your database. This command file must specify two kinds of information:

- One or more **FILE** statements, to define data fields within the records of the input file(s)
- One or more **INSERT** statements, to indicate how to place the new data into the columns of the database table(s)

**INSERT** statements in **dbload** command files resemble **INSERT** statements in **SQL**, except that they cannot incorporate **SELECT** statements (since the data are not yet in any table). In effect, the most recent **FILE** statement replaces **SELECT** in defining the list of data values for a **dbload INSERT** statement to enter as new rows.

The format of a command file for **dbload** is indicated here:

---

```
FILE { "filename" }  
  
    { DELIMITER "c" nfields |  
  
      (fieldn1 start [- end ][: ... ] [ NULL = "null-str1" ],  
        fieldn2 start [- end ][: ... ] [ NULL = "null-str2" ],  
  
        fieldnN start [- end ][: ... ] [ NULL = "null-strN" ] ) } ;  
  
INSERT INTO tablename [ (column-list) ] [ VALUES (value-list) ] ;  
  
[ . . . ]
```

---

The command file can include multiple **FILE** and **INSERT** statements. An explanation of these terms, notes, and an example follow.

## Explanation

<b>FILE</b>	is a required keyword.
<i>filename</i>	is the pathname of an input file, enclosed between a pair of quotation ( " ) marks.
<b>DELIMITER</b>	is a keyword that is required if <i>filename</i> has variable-length data records.
<i>c</i>	is the field delimiter (enclosed in quotes) between fields of a variable-length data record, and before the <b>NEWLINE</b> character that terminates each record.
<i>nfields</i>	is an integer, specifying the number of fields in each variable-length data record.
<i>fieldn</i>	is a name that you assign to a data field within a fixed-length record of <i>filename</i> .
<i>start</i>	is an integer, indicating a character position within a fixed-length record.
<i>-end</i>	is a hyphen ( - ) and an integer, indicating (with <i>start</i> ) a range of character positions.
<b>NULL</b>	is a keyword to specify a <b>NULL</b> symbol.
<i>null-str</i>	is a quoted string, specifying a data value for which <b>dbload</b> should substitute a <b>NULL</b> .
<b>INSERT INTO</b>	are required keywords.
<i>tablename</i>	identifies a table in which to store the data.
<i>column-list</i>	is a list of column names within <i>tablename</i> , separated by commas.
<b>VALUES</b>	is an optional keyword to specify a list that can include constants and data field names.
<i>value-list</i>	is a comma-separated list of constants and/or data field names from <i>filename</i> .

## Notes

1. The **dbload** utility recognizes valid *owner.table* references.
2. You need UNIX *read* permission for *filename*, and you must also be granted the **INSERT** privilege for *tablename*.
3. Every statement must end with a semicolon ( ; ) symbol.
4. You cannot specify character positions or *null-str* symbols in a record defined with the **DELIMITER** option.
5. Use a colon ( : ) to separate character position or range values in each data field definition. The list of field definitions must be enclosed in parentheses and separated by commas.
6. The same character position can be repeated in the **FILE** specification of a field, or in different fields. (See the command file example that follows these notes.)
7. The scope of reference of a *null-str* is the field for which you define it, but you can define the same *null-str* for other fields.
8. The **DELIMITER** option automatically assigns the sequential names **f01**, **f02**, **f03**, . . . to fields in variable-length records. The *value-list* of an **INSERT** statement can reference field names assigned by the user or by **dbload** in the previous **FILE** statement.
9. If your **INSERT** statement omits the *column-list*, then the default columns are every column in *tablename*. If you do not specify a *value-list*, then the default values are those in every field of the previous **FILE** statement.
10. An error results if the *column-list* and the *value-list* have different numbers of elements.
11. If the *column-list* includes fewer columns than *tablename*, **dbload** attempts to insert **NULL** values in the remaining columns. If a **NOT NULL** restriction or **UNIQUE CONSTRAINT** would be violated, the insertion fails, and an error message appears.
12. Inserted data types correspond to the explicit or default *column-list*. If the data field width is different from its corresponding character column, inserted values are padded with blanks if the column is wider, or are truncated if the field is wider.
13. Enclose between braces ( { } ) any comments in *filename*.



14. Use the **DELIMITER** option to avoid truncation of long character fields. If the delimiter *c* (or a backslash) appears as a literal character, you must prefix it with a backslash ( \ ) in the input file.
15. If you specify **DELIMITER**, the same delimiter must be used throughout the input file and must appear in quotes in the **FILE** statement. You must remember to place the delimiter immediately before the **NEWLINE** character that marks the end of each record. (If you omit this delimiter, an error results whenever the last field of a record is empty.)

## Examples

---

```
FILE "datafile1" (fld1 1 - 10 : 13 : 5 - 22  NULL = "str1" ,
                fld2 10 - 21 : 28 - 32 ,
                fld3 8 - 10 : 33 - 50 : 29 - 33  NULL = "str2" ,
                . . .
                fldN 9 : 16 - 19  NULL = "string") ;

INSERT INTO tab1 (col1, col2, col9, . . . , colN) ;

INSERT INTO tab2
VALUES  (fld1, fld3, "kevin", . . . , fldN) ;

INSERT INTO tab3 ;           {no column or values list provided}

FILE "datafile.2" DELIMITER "|" 8 ;  {variable-length fields}

INSERT INTO tab1
VALUES  (f01, f02, "kevin", "234", . . . , f08) ;

INSERT INTO tab4 ;
```

---

**Note:** The ellipses ( . . . ) in this example are typographic conventions that cannot appear in command files. Unless you use the **DELIMITER** option, for example, you must explicitly list every field that a **FILE** statement defines. Each statement in this example is described in the pages that follow.

```
FILE "datafile1" (fld1 1 - 10 : 13 : 5 - 22  NULL = "str1" ,
```

Here *datafile1* is the input file, and *fld1*, *fld2*, *fld3*, through *fldN* are user-assigned field names in its fixed-length data records. In this example, *fld1* consists of the characters in positions 1 through 10, 13, and 5 through 22 of every *datafile1* record. (Each record ends with a **NEWLINE** character.) Notice that the characters 5 through 10 and 13 appear twice in *fld1*, and characters 10, 13, and 21 appear in *fld1* and *fld2*.



In field *fld1*, the NULL symbol is defined as “str1.” A NULL value is entered whenever “str1” is read in *fld1*.

The *fld2* field consists of positions 10 through 21, and 28 through 32.

The *fld3* field is defined as the characters in positions 8 through 10, 33 through 50, and 29 through 33. The NULL symbol for field *fld3* is defined as “str2.”

The field-definition process continues until the last field is reached. Field *fldN* contains characters in positions 9, and 16 through 19. The NULL value is defined as “string.”

```
INSERT INTO tab1 (col1, col2, col9, . . . , colN) ;
```

An INSERT statement follows. Here *col1*, *col2*, *col9*, and so on are the actual database column names in table *tab1*. Since no value list is provided, **dbload** takes the values in the fields defined in the preceding FILE statement. It inserts the data from *fld1* into *col1*, from *fld2* into *col2*, from *fld3* into *col9*, and so forth, until the value from *fldN* is inserted into *colN*. (Columns 4 through 8 are skipped, so the new rows will have NULL values there, if the columns permit NULLs.)

```
INSERT INTO tab2  
VALUES (fld1, fld3, "kevin", . . . , fldN) ;
```

Since no column list is provided, **dbload** reads the names of all the columns in *tab2* from the system catalogs. Values to load into each column are specified by field names from the previous FILE statement or as constants. Data in *fld1* go into the first column, data from *fld3* into the second, and the constant “kevin” into the third. The **dbload** utility continues until the value in *fldN* is inserted into the final column.

```
INSERT INTO tab3 ;      {no column or values list provided}
```

As noted in the comment, this statement specifies no column names or data values. To create a default *column-list*, **dbload** checks the system catalogs for the names of all the columns in table *tab3*.

The default *value-list* comes from the most recent FILE statement, in this case the first statement in the command file. Data from *fld1* go into the first column, data from *fld2* into the second, and so forth, with data from field *fldN* going into the *N*th column.

**Note:** This statement requires that the field list in the **FILE** statement have a one-to-one correspondence with the columns of table **tab3**, as listed in the system catalogs. Unless this correspondence exists, **dbload** will not load the records. Instead, you will receive an error message on each record.

The loading process terminates when the total number of records that cannot be inserted as new rows exceeds a limit that you can specify at the **dbload** command line (by using the **-c** option).

```
FILE "datafile.2" DELIMITER "|" 8 ; {variable-length fields}
```

The **DELIMITER** clause tells **dbload** that file **datafile.2** has variable-length fields, and that the vertical-bar character ( **|** = ASCII 124) separates each field.

Here 8 is the number of fields in each input record. Fields are automatically assigned the names *f01*, *f02*, *f03*, and so on.

```
INSERT INTO tab1  
VALUES (f01, f02, "kevin", "234", . . . , f08) ;
```

A *value-list* but no *column-list* is specified, so **dbload** reads all the column names of **tab1** from the system catalogs. Here the value in field *f01* goes into the first column, the *f02* value into the second, the constant "kevin" into the third, the constant "234" into the fourth, and so forth, until the value in field *f08* is inserted into the last column.

You must reference fields in a variable-length data file with the letter **f** followed by a two-digit number: *f01*, *f02*, *f10*, and so on. Other formats like *fld01* or *f3* are incorrect.

```
INSERT INTO tab4 ;
```

Since no *column-list* or *value-list* is provided, **dbload** finds all the names of columns in table **tab4** in the system catalogs. The *value-list* is all the fields defined in the previous **FILE** statement. (Notice that this is not the same **FILE** statement that was used with table **tab3**.)

If these values have a one-to-one correspondence with the columns, the value from field *f01* goes into the first column, the value from *f02* into the second column, and so on, until the value in *f08* is placed in the last column. An error results if 8 is not the number of columns in table **tab4**.

## Specifying a dbload Command Line

After you have created a valid command file, you invoke **dbload** at the operating system prompt. The argument list that you include in your command line determines whether the program operates in *command mode* or *interactive mode*.

If you enter the keyword **dbload** without any arguments, the screen displays a syntax summary for **dbload** usage, and control returns to the operating system.

To use **dbload** to read and execute a command file, you must enter a command line that includes at least one of its required specifications. The following elements are required in a **dbload** command line:

---

```
dbload { -d database | -c comfile | -l logfile } . . .
```

---

- |                           |  |
|---------------------------|--|
| <b>dbload</b>             | invokes the <b>dbload</b> utility.             |
| <b>-d</b> <i>database</i> | identifies a database to receive the new data. |
| <b>-c</b> <i>comfile</i>  | identifies a <b>dbload</b> command file.       |
| <b>-l</b> <i>logfile</i>  | identifies a file to log any error messages.   |

The following sections describe both the interactive and command modes of **dbload**.



## *Running dbload Interactively*

If you specify part (but not all) of the required information after the **dbload** keyword, the program automatically enters interactive mode and prompts you for additional specifications. Depending on what you entered at the command line, these specifications can include

- The name of the database to receive the new data.
- The name of the command file to be executed.
- The name of an error log file in which to store any input file records that **dbload** cannot insert into the database, as well as diagnostic information.
- Whether you only want **dbload** to check the syntax of the **FILE** and **INSERT** statements in your command file, without changing the database.
- How many bad input records can be encountered before **dbload** stops inserting new rows.
- How many input file records to read before committing new rows to the database, if your database supports transactions.
- Whether to discard or to commit to the database any rows that were successfully read between the last **COMMIT** and the first bad record that exceeds your cumulative error limit.
- How many input records to ignore before **dbload** begins to insert data. (That is, how many **NEWLINE** characters to read before actual processing begins.)

After you enter these specifications or press **RETURN** to accept the default values that appear in the prompts, the screen is cleared, and the **dbload** utility begins execution.



## Running dbload in Command Mode

If a valid **dbload** command line includes the required *database*, *command file*, and *error log* specifications, with or without any additional options, program execution begins. The complete syntax of **dbload** is indicated here:

### Syntax

---

```
dbload  -d database  -c comfile  -l logfile  
        [-e num1 ] [-n num2 ] [-i num3 ] [-p ] [-r ] [-s [ > outfile ] ]
```

---

### Explanation

**dbload** is a required keyword.

**-d *database*** is the name of a database to receive the data.

**-c *comfile*** is the pathname of a **dbload** command file.

**-l *logfile*** is the pathname of an error logging file.

**-e *num1*** is the number of bad records that **dbload** will read before it terminates (where *num1* is an integer).

**-n *num2*** displays a message after each batch of *num2* new rows are inserted (where *num2* is an integer).

**-i *num3*** ignores the first *num3* input records (where *num3* is an integer).

**-p** prompts for instructions if the number of bad records exceeds *num1*.

**-r** instructs **dbload** not to lock the table(s). (Otherwise, table-level locking occurs during loading.)

**-s** instructs **dbload** only to check the syntax of the statements in *comfile*, without inserting any data.

**> *outfile*** is an optional > symbol and the name of a file in which to save output from the syntax check.

## Notes

1. Unless you include at least one of the first three options, **dbload** displays a help message and terminates. If you omit one or two of the three required options, **dbload** prompts you for additional specifications.
2. You can prefix *comfile*, *logfile*, or *outfile* with a pathname.
3. You should run **dbload** with the **-s** or **-s > outfile** options before you begin loading data. These options perform a syntax check on the **FILE** and **INSERT** statements in *comfile* and ignore any other options except **-d database** and **-c comfile**. The screen displays *comfile* with any errors marked where they are found.
4. If you do not specify a value for *num1*, the default is 10 bad records, so the program stops loading when it reads the 11th bad record. If you set *num1* at zero, **dbload** terminates when it reads the first bad record.
5. If your database supports transactions, **dbload** reads and inserts a batch of *num2* records between each **COMMIT**. If you do not specify a value for *num2*, the default is 100 records.
6. The **-i** option instructs **dbload** to ignore the first *num3* lines in the input file, so processing does not begin until **dbload** has read *num3* NEWLINE characters. This option is useful, for example, if your most recent **dbload** session with the same command file ended after 240 lines of input. You can resume loading at line 241 by setting *num3* equal to 240. It is also useful if header information in the input file precedes the data records.
7. After (*num1* + 1) bad records in a database with transactions, the **-p** option prompts you to roll back or to commit any rows inserted since the last transaction. The default is to commit.
8. If you press the Interrupt key, **dbload** terminates and discards any new rows that have been inserted but not yet committed to the database (if the database has transactions).
9. The presence of indexes greatly affects the speed with which the **dbload** utility loads data. For best performance, drop any indexes on the tables receiving the data before you run **dbload**. You can create new indexes after **dbload** has finished.

## Examples

The following example shows a **dbload** command line at the system prompt that uses all options (except the **-s** option):

---

```
dbload -d stores -c cfyl -l lfyl -e 5 -n 75 -i 20 -p -r
```

---

This example specifies the following parameters:

- d stores      is the name of the database to receive the data.
- c cfyl        is the name of the **dbload** command file.
- l lfyl        is the name of the error-log file.
- e 5           specifies that **dbload** will log up to five bad records. The program terminates if a 6th bad record is encountered.
- n 75          specifies that screen messages will indicate when successive batches of 75 rows have been inserted (or committed, in a database with transactions).
- i 20          tells **dbload** to ignore the first 20 lines of the data file. Processing begins at the 21st line.
- p            prompts you to commit or discard any uncommitted rows after six bad records.
- r            tells **dbload** not to lock the table(s) of the **stores** database that are being accessed by **dbload** while the new rows are being inserted. This allows other users to access the same table(s) during the **dbload** operation. Unless you specify the **-r** option, table-level locking occurs.

Notice that the names of the input file and the table(s) to receive the data do not appear explicitly in the command line. These names must be specified within the **dbload** command file, which is called **cfyl** in this example.



# The *dbschema* Utility

You can use the **dbschema** utility to quickly produce an SQL command file containing the CREATE TABLE, CREATE INDEX, and CREATE VIEW statements required to replicate an entire database or a selected table. In addition, **dbschema** can produce all CREATE SYNONYM and GRANT statements in effect for the database or a selected table or view. You must be the DBA or have CONNECT or RESOURCE permission to the database before you can run **dbschema**.

By default, **dbschema** produces all CREATE TABLE, CREATE VIEW, CREATE INDEX, CREATE SYNONYM, and GRANT statements in effect for the entire database. By including the appropriate command line option, you can limit the output to a selected table or view, or produce synonyms and permissions for a particular user.

The **dbschema** utility uses the *owner.object* convention when it generates any CREATE TABLE, CREATE INDEX, CREATE SYNONYM, CREATE VIEW, or GRANT statements, and when it reproduces any UNIQUE CONSTRAINTs. As a result, if you use the **dbschema** output to create a new object (table, index, synonym, or view), the new object is owned by the owner of the original object. If you want to change the owner of the new object, you must edit the **dbschema** output before running it as an SQL script.

The syntax for the **dbschema** utility follows:

---

```
dbschema [-t tablename] [-s sname] [-p pname] -d database [filename]
```

---

**-t *tablename*** specifies the table or view for which you want **dbschema** to output CREATE TABLE and CREATE INDEX statements or the CREATE VIEW statement. If you specify all for *tablename*, **dbschema** outputs the SQL statements for all database tables and views.

**-s *sname*** specifies the user for whom you want **dbschema** to output the CREATE SYNONYM statements. If you specify all for *sname*, **dbschema** outputs all CREATE SYNONYM statements. If you include the



**-t option, *dbschema* produces the CREATE SYNONYM statements only for the indicated *tablename*.**

**-p *pname*** specifies the user for whom you want ***dbschema*** to output permission statements. If you specify all for *pname*, ***dbschema*** outputs the GRANT statements for all users. If you include the **-t** option, ***dbschema*** produces the GRANT statements only for the indicated *tablename*.

**-d *database*** specifies the name of the database.

***filename*** specifies the name of the file in which to save the ***dbschema*** output. If *filename* is not provided, ***dbschema*** outputs to the screen.

## Notes

1. The following command line produces the SQL statements necessary to replicate an entire database:

```
dbschema -d database
```

where *database* is the name of the database.

2. You must be the DBA or have CONNECT or RESOURCE permission to the database before you can run ***dbschema*** on it.
3. When you include the **-t** option, ***dbschema*** produces SQL statements only for the indicated *tablename*. The ***dbschema*** utility uses the *tablename* to filter the output. If you use the **-t** option with the **-s** and **-p** options, only the CREATE SYNONYM and GRANT statements for *tablename* are provided.
4. All objects listed in the ***dbschema*** output include the name of the owner. If you want to use the ***dbschema*** output to create a schema with different owners, you must first change the owner names in the output file.
5. All SERIAL fields included in CREATE TABLE statements output by ***dbschema*** have a starting value of one, regardless of their original starting value.

6. In the **dbschema** output, the **AS** keyword is used to indicate the grantor of a **GRANT** statement, as in the following command:

```
grant all on "tom".customer to "claire" as "norma"
```

This statement tells you that *norma* issued the **GRANT** statement.

7. When the **GRANT..AS** keywords appear in your **dbschema** output, you may need to grant certain permissions before running the output as an SQL script. Consider the following **GRANT** statement:

---

```
grant tab-privilege on user1.tablename to "user2" as "user3"
```

---

Before this statement can be run to create another schema, the following must be true:

- *user3* must have **CONNECT** permission to the database.
  - *user3* must have *tab-privilege* **WITH GRANT OPTION** for *tablename*.
8. The *database* must exist in your current directory or in a directory cited in your **DBPATH** environment variable.

## Examples

The following statement outputs the SQL statements relating to the **customer** table in the **stores** database to the file **c\_schema.sql**.

---

```
dbschema -t customer -s alice -p dinah -d stores c_schema.sql
```

---

The output consists of the following components:

- The **CREATE TABLE** and **CREATE INDEX** statements for the **customer** table
- All **CREATE SYNONYM** statements executed by the user *alice* on the **customer** table
- All permissions granted to the user *dinah* on the **customer** table

The output from this **dbschema** statement follows:

---

```
{ TABLE "alice".customer row size = 134 number of columns = 10 index size = 0 }
create table "alice".customer
(
    customer_num serial not null,
    fname char(15),
    lname char(15),
    company char(20),
    address1 char(20),
    address2 char(20),
    city char(15),
    state char(2),
    zipcode char(5),
    phone char(18)
);
revoke all on "alice".customer from "public";

create unique index "alice".c_num_ix on "alice".customer (customer_num);
create index "alice".zip_ix on "alice".customer (zipcode);

grant all on "alice".customer to "dinah" as "alice";

create synonym "alice".cust for "alice".customer;
```

---

The next **dbschema** statement outputs the SQL statements for all tables in the **stores** database to the file **s\_schema.sql**.

---

```
dbschema -t all -s alice -p alice -d stores s_schema.sql
```

---

The output consists of the following components:

- The **CREATE TABLE**, **CREATE VIEW**, and **CREATE INDEX** statements that replicate all tables, views, and indexes in the **stores** database
- All **CREATE SYNONYM** statements executed by the user *alice*
- All permissions granted to the user *alice*



# The *dbupdate* Utility

Databases created through Informix products that used an early implementation of SQL (Version 1.1 or earlier) have a different structure than databases created through products using the current implementation. The current structure includes additional system catalogs, content changes to some existing system catalogs (see Appendix B), and the introduction of NULL values.

A database will have the old structure if it was created by an application using Version 2.0 (or earlier) of **INFORMIX-SQL**, **INFORMIX-ESQL/C**, or **INFORMIX-ESQL/COBOL**, or Version 1.0 of **INFORMIX-4GL**. Such a database cannot be used with current application development tools or database engines until the database is converted to the current structure. You can convert old databases to the current structure through the **dbupdate** utility.

## Using *dbupdate*

To convert an old database to the current structure, execute the following command:

---

```
dbupdate [-b | -n] old-database-name new-database-name
```

---

The **dbupdate** utility creates a new database in the current directory with the name *new-database-name*, and copies the data from the old system catalogs to the new system catalogs, making the appropriate changes.

If you do not use the **-b** or **-n** option, **dbupdate** converts the value of all CHAR type columns with blank data to NULL and, for each numeric column, asks you if it should convert zero values to NULL values.

The **-b** option causes **dbupdate** to leave blank data in CHAR columns as blanks. The **-n** option alters the system catalogs to define all columns as NOT NULL and does not touch the data files. The **-n** option includes the **-b** option.



In addition to these changes, **dbupdate** corrects a bug in the representation of negative DECIMAL values.

When **dbupdate** finishes, you have two database directories (the new and the old) with two separate system catalogs, but the data and index files are shared (linked). To complete the update, remove the old database directory.

## ***No NULL Databases***

You may want to avoid working with NULL values. You can do this if you carefully adhere to the following rules:

- When converting an old database, select the **-n** option of **dbupdate**.
- When creating new tables, define all columns as NOT NULL.
- In all form specification files, add the WITHOUT NULL INPUT clause in the DATABASE section.
- In form specification files, specify all **formonly** fields as NOT NULL.

These last two rules mean you must recompile all your old form specification files.

# The *sqlconv* Utility

The **sqlconv** utility is provided for the Database Administrator who wants to use **INFORMIX-SQL**, **INFORMIX-ESQL/C**, **INFORMIX-4GL**, or other application development tools with a database that was created with the non-SQL product **INFORMIX** (Version 3.0 or higher). From the **INFORMIX** database, **sqlconv** helps a user create a new SQL-compatible database. It leaves the old database intact.

This appendix discusses the steps necessary to convert an **INFORMIX** database to an SQL-compatible database. The new database can be used with any Informix application development tool.

**Note:** If you have purchased another application development tool in addition to **INFORMIX-ESQL/C**, and you have already run the **sqlconv** utility provided with the other product, you do not need to run **sqlconv** again.

This appendix describes two methods you can use to accomplish a database conversion. The method you should use depends on your system's disk space constraints.

---

**Caution!** **sqlconv** will not convert **INFORMIX** database permissions. You must grant new permissions on tables and fields after your new **INFORMIX-ESQL/C** database has been created and loaded.

SQL reserved words are not the same as **INFORMIX** reserved words. If you receive a syntax error while running your new **CREATE** scripts or **INFORMIX-ESQL/C** programs, make sure your **TABLE** and **FIELD** names are not among the new reserved words. For a list of reserved words, see Appendix D, "Reserved Words," in this manual.

Make sure all **INFORMIX** composite fields are indexed. Indexed composite fields will have composite indexes created for them.

If you have used the **LOCATION** option to spread an **INFORMIX** database across a number of directories, converting the database using **sqlconv** places all of these files in the new **.dbs** directory.

You cannot specify a new starting number for a serial field.

---

# Conversion Procedures

## If There Is No Shortage of Disk Space

You can convert an entire **INFORMIX** database to an SQL-compatible database for use with **INFORMIX-ESQL/C** at one time if there is no shortage of disk space.

To convert an entire database at once, you must have available on the disk at least three times the amount of space required by all the tables in the database plus additional space for the **INFORMIX-ESQL/C** system files. The following steps outline the process of converting an entire database at once:

1. Make certain that **INFORMIX** and **INFORMIX-ESQL/C** are included in your search path.
2. Set the **INFORMIXDIR** environment variable to point to the **INFORMIX-ESQL/C** directory.
3. Make your current directory the directory that contains your **INFORMIX** database.
4. Create a backup copy of the **INFORMIX** database.
5. Enter

```
sqlconv -e databasename
```

where *databasename* is the name of the **INFORMIX** database you want to convert. Do not include a filename extension. This command generates an **INFORMER** script file (indicated by a **.uld** extension), an **INFORMIX-ESQL/C** program (indicated by a **.ec** extension), and a **dbload** command file (indicated by a **.cmd** extension).

6. Enter

```
informer databasename databasename.uld
```

This command unloads the database files (in ASCII format) to *unload* files (indicated by a **.unl** extension) for each table in the database.



7. Enter

```
esql databasename.ec -o databasename.out
```

This command compiles the INFORMIX-ESQL/C program created in Step 5, and creates an *.c* file (a C program) and an *.out* executable file.

8. Enter

```
databasename.out
```

This command runs the INFORMIX-ESQL/C program that you compiled in Step 7 and recreates the database, tables, and indexes in SQL format.

9. Enter

```
dbload -d databasename -c databasename.cmd -l errlog
```

This command loads the data from the *.unl* files (generated by **informer**) into the appropriate tables and creates an *errlog* file, which contains diagnostic information about any rows that were not successfully loaded. For more information on **dbload**, see the discussion of "The *dbload* Utility" in this appendix.

10. The final step in the conversion procedure is to remove all the old database files, the *.uld*, *.ec*, *.cmd*, *.unl*, *.c*, and *.out* files from your directory. Do not remove your forms and reports. You can update these later.

## If There Is a Shortage of Disk Space

The following method is more economical in terms of disk space, but it is a more involved and time-consuming process than the method discussed in the previous section. Use this method if you do not have at least three times the amount of space required by the database.

The following steps outline the conversion of the sample **INFORMIX** database, **payroll**, to an SQL-compatible database for use with **INFORMIX-ESQL/C**:

1. Make certain that **INFORMIX** and **INFORMIX-ESQL/C** are included in your search path.



2. Set the **INFORMIXDIR** environment variable to point to the **INFORMIX-ESQL/C** directory.
3. Make your current directory the directory that contains your **INFORMIX** database.
4. Create a backup copy of the **INFORMIX** database.
5. Enter

```
sqlconv -e payroll
```

Do not include a filename extension. This command generates an **INFORMER** script file (indicated by a **.uld** extension), an **INFORMIX-ESQL/C** program (indicated by a **.ec** extension), and a **dbload** command file (indicated by a **.cmd** extension).

6. Enter

```
esql payroll.ec -o payroll.out
```

This command compiles the **INFORMIX-ESQL/C** program you created in Step 5 and creates a **.c** file (a C program) and an **.out** executable file.

7. Enter

```
payroll.out
```

This command runs the **INFORMIX-ESQL/C** program you compiled in Step 6 and recreates the database, tables, and indexes in **SQL** format, but it does not load the data.

8. Enter

```
cat payroll.uld
```

The statements in the **payroll.uld** file outline the first steps of the conversion operation. You must execute each of these statements separately. You may find it helpful to print a copy of the **payroll.uld** file for easy reference.

9. Enter

informer payroll

At the **INFORMER** prompt, enter the first line of the **payroll.uld** file exactly as it appears. This creates the unload file for the first table. Exit **INFORMER**.

10. The **.cmd** file describes the form of the data and contains the **INSERT INTO** statements indicating how this data is to be placed in the database files. The **INSERT INTO** statements are necessary to load the data into the newly created database. You must execute each of the statements separately. To do this, create a copy of the **payroll.cmd** file for each **INSERT INTO** statement and make sure you include the **.cmd** extension to the filename of each new file. In this instance, we have named the files **one.cmd** and **two.cmd**.
11. Edit the **one.cmd** file using your system editor and remove all but the first **INSERT INTO** statement from the file. Exit the file.
12. Execute the first **INSERT INTO** statement in the **one.cmd** file by entering

```
dbload -d payroll -c one.cmd -l errlog
```

This command loads the data from the **.unl** file into the appropriate table and creates an *errlog* file, which contains diagnostic information about any rows that were not successfully loaded. A statement appears on the screen indicating how many rows were loaded into the file. For more information on **dbload**, see the discussion of “The *dbload* Utility” in this appendix.

13. Before you can perform the same operations on any other database tables, you must free additional disk space by erasing the **INFORMIX** versions of the recently created **INFORMIX-ESQL/C** files. To erase these files, enter

```
dbstatus payroll
```

14. At the **dbstatus** prompt, enter

erase file *filename*

where *filename* is the name of the file referred to in the **.cmd** file created in Step 11. Exit **dbstatus**.

15. Erase the unload file for this same file by entering from the command line

rm *filename.unl*

16. Continue to unload each table, one at a time, from the **INFORMIX** database and then load it into the **INFORMIX-ESQL/C** database. Do this by repeating Steps 9 through 15. Remember to make a copy of the **.cmd** file for each **INSERT INTO** statement it contains. Each copy must have a unique name and must end in the **.cmd** extension. The correct filename must be included on the command line each time you run the **dbload** command.

Repeat these steps until all load statements in the **payroll.cmd** file have been executed. This operation loads the actual data into the newly created database.

17. Check the contents of the data files in the newly created database to make sure you have successfully converted your **INFORMIX** database.
18. When all tables in the database have been converted, erase the **.uld**, **.ec**, **.cmd**, **.c**, and **.out** files, and drop the **INFORMIX** database.

---

**Caution!** You cannot rerun **sqlconv** after you have erased a table. The new scripts generated by the command do not contain the information necessary to convert the table.

---

# The *dbexport* Utility

## Overview

Use **dbexport** to unload a database into ASCII files for import into another database environment.

## Syntax

---

```
dbexport [-c] [-q] database  
          [-o directory-path | -t device -b blksize -s tapesize [-f pathname]]
```

---

## Explanation

<b>dbexport</b>	is the program name.
<b>-c</b>	tells the program to continue even though errors occur.
<b>-q</b>	tells the program not to display anything on its standard output.
<i>database</i>	is the name of the database to be exported.
<b>-o <i>directory-path</i></b>	directs the output to a particular directory on disk.
<b>-t <i>device</i></b>	directs the output to a particular tape device.
<b>-b <i>blksize</i></b>	specifies the tape block size in kilobytes.
<b>-s <i>tapesize</i></b>	specifies the capacity of one tape reel.
<b>-f <i>pathname</i></b>	tells the program to write data definition statements to the file <i>pathname</i> and not to the tape.



## Notes

1. You must have DBA privilege or log in as user **informix** to export a database.
2. The database is locked in exclusive mode during export. If an exclusive lock cannot be obtained, the program ends with a diagnostic message.
3. The **dbexport** program always creates a file of messages called **dbexport.out**. This file contains any error messages and warnings, and it also contains a display of the SQL data definition statements it is generating. The same material is also written to the standard output unless you specify the **-q** option.
4. You can cancel the program with an interrupt signal. The **dbexport** program asks for confirmation before terminating.
5. The **dbexport** program writes multiple files containing database data, either to disk or to tape. The **-t** option specifies that the destination is a tape drive; otherwise, **dbexport** writes the files to disk. When you include the **-t** option, you must also specify the tape device, the block size, and the volume capacity.
6. When you include the **-t** option, the file of data definition statements and other commands (used by the **dbimport** utility) are ordinarily also written to the tape. Use the **-f** option to instruct the program to write these to the file *pathname*. This allows you to inspect and modify the statements.
7. When you do not include the **-t** option, the destination is a disk directory with the name *database.exp*. This directory must not exist; the program will create it. Its group will be **informix**. If you include the **-o directory-path** option, the *database.exp* directory is located in the specified directory. By default, the database is placed in your current working directory.
8. When output is to disk, the file containing the data definition statements and other commands to **dbimport** is written to the file *database.sql* in the *database.exp* directory.

## Examples

The following command exports the **stores** database to tape with a block size of 16 kilobytes and a tape capacity of 24,000 kilobytes. The file of data definition statements and other directions to **dbimport** is written to **stores.imp** in the **/tmp** directory.

```
dbexport -c stores -t /dev/rmt0 -b 16 -s 24000 -f /tmp/stores.imp
```

The following command exports the **stores** database to the **/usr/informix/port/stores.exp** directory.

```
dbexport -c stores -o /usr/informix/port
```

# The *dbimport* Utility

## Overview

Use **dbimport** to create a database from ASCII files.

## Syntax

---

```
dbimport [-c] [-q] database  
          [-i directory-path | -t device -b blksize -s tapesize [-f pathname] ]  
          [-d dbspace] [-l [ logpath | buffered] ] [-ansi]
```

---

## Explanation

- |                                 |   |
|---------------------------------|---|
| <b>-c</b>                       | tells the program to continue even when errors occur, unless it is a fatal error.                         |
| <b>-q</b>                       | tells the program not to display anything on its standard output.   |
| <i>database</i>                 | is the name of the database to import.  |
| <b>-i</b> <i>directory-path</i> | specifies the path to an input directory.   |
| <b>-t</b> <i>device</i>         | specifies input from a particular tape device.  |
| <b>-b</b> <i>blksize</i>        | specifies the tape block size in kilobytes.   |
| <b>-s</b> <i>tapesize</i>       | specifies the capacity of one tape reel.  |
| <b>-f</b> <i>pathname</i>       | tells the program to read data definition statements from the file <i>pathname</i> and not from the tape. |
| <b>-d</b> <i>dbspace</i>        | when importing to <b>INFORMIX-OnLine</b> only, specifies the dbspace where the new database is to go.     |
| <b>-l</b>                       | specifies that the imported database is to use transaction logging.                                       |

<i>logpath</i>	when importing to INFORMIX-SE only, specifies the pathname of the transaction log file.
<b>buffered</b>	when importing to INFORMIX-OnLine only, specifies buffered or unbuffered logging (unbuffered is the default).
<b>-ansi</b>	tells the program to create the database as MODE ANSI.

## Notes

1. The program always creates a message file called **dbimport.out** in the current directory. This file contains messages and warnings related to the running of the program. The messages are also written to the standard output (normally the terminal screen) unless you include the **-q** option.
2. You can cancel the program with an interrupt signal. You are prompted for confirmation before the program terminates.
3. The individual who runs **dbimport** is granted DBA privilege on the new database.
4. When importing a database that uses INFORMIX-SE, database files are created in the current directory.
5. Use the **-l** option to establish transaction logging for the imported database. This option is equivalent to the WITH LOG IN clause of the CREATE DATABASE statement. A database created as MODE ANSI requires transaction logging. In this situation, you must include the **-l** option.
6. The **dbimport** utility reads multiple files containing database data from either disk or tape. Use the **-t** option to specify the source as tape; the default is disk. When you include the **-t** option, you must also specify the tape device, block size, and volume capacity.
7. When you include the **-t** option, **dbimport** reads the data definition statements and other **dbimport** commands from the tape. Use the **-f pathname** option to instruct the program to read the *database.sql* file in *pathname* (instead of the tape) for the data definition statements and other commands.



To use the **-f** option you must have also used it when you executed the **dbexport** program.

8. If you do not specify the **-t** option, the source of the database data is a disk directory with the name *database.exp*. The **dbimport** program looks for this directory in the current working directory, or on the path specified with the **-i** option. In either case, the program takes data definition and other commands from the file *database.sql* in the directory *database.exp*. (This is why the name *database* must be the same as was given to **dbexport**.)

## Examples

The following command imports the **stores** database from a tape with a block size of 16 kilobytes and capacity of 24,000 kilobytes. The file of data definition statements and other import commands was put in **stores.imp** in the **/tmp** directory when **dbexport** was run.

```
dbimport -c stores -t /dev/rmt0 -b 16 -s 24000 -f /tmp/stores.imp
```

The following command imports the **stores** database from the **/usr/informix/port/stores.exp** directory using data definition and commands from the **stores.sql** file in that directory. The new database is created as **MODE ANSI** and uses logging.

```
dbimport -c stores -i /usr/informix/port -ansi -l /usr/work/stores.log
```

# **Appendix G**

## **Outer Joins**



## Outer Joins

This appendix discusses the difference between a simple join and an outer join and describes in detail how outer joins work. The following SELECT statements illustrate the basic difference between the two types of join.

### Query 1 Using a simple join

---

```
SELECT customer.customer_num, lname, order_num
      FROM customer, orders
      WHERE customer.customer_num = orders.customer_num
```

---

### Query 2 Using an outer join

---

```
SELECT customer.customer_num, lname, order_num
      FROM customer, OUTER orders
      WHERE customer.customer_num = orders.customer_num
```

---

Both query the same tables (**customer** and **orders**) of the same database (**stores**) through a join on the same column (**customer\_num**). At first glance, both fetch the same data. The query results, however, are quite different.



### Query 1 Results

customer_num	lname
104	Higgins
101	Pauli
104	Higgins
106	Watson
116	Parmelee
112	Lawson
117	Sipes
110	Jaeger
111	Keyes
115	Grant
104	Higgins
117	Sipes
104	Higgins
106	Watson
110	Jaeger

### Query 2 Results

order_num	customer_num	lname	order_num
1001	101	Pauli	1002
1002	102	Sadler	
1003	103	Currie	
1004	104	Higgins	1001
1005	104	Higgins	1003
1006	104	Higgins	1011
1007	104	Higgins	1013
1008	105	Vector	
1009	106	Watson	1004
1010	106	Watson	1014
1011	107	Ream	
1012	108	Quinn	
1013	109	Miller	
1014	110	Jaeger	1008
1015	110	Jaeger	1015
	111	Keyes	1009
	112	Lawson	1006
	113	Beatty	
	114	Albertson	
	115	Grant	1010
	116	Parmelee	1005
	117	Sipes	1007
	117	Sipes	1012
	118	Baxter	

Query 1 fetches a list of only those customers who have items on order by using a *simple join*, while Query 2 fetches a list of all customers by using an *outer join*. Once you understand how similar queries can produce such dissimilar results, you can begin to use outer joins effectively. The obvious differences between the two kinds of joins are as follows:

- A *simple join* discards all rows that do not satisfy the join condition.
- An *outer join* preserves rows that would otherwise be discarded.

The following section discusses outer joins in detail.

## How Outer Joins Work

A join queries two or more tables as though they were one. It is as if **INFORMIX-ESQL/C** creates and then acts upon a single temporary table to produce the query results. **INFORMIX-ESQL/C** does not actually create such a table to perform a join, but it is helpful to conceptualize a join in these terms.

In a simple two-table join, the resulting “table” contains only those combinations of rows from both tables that satisfy the join condition. In an outer join, the resulting “table” contains these rows plus all remaining rows from one of the tables, called the dominant (or preserved) table. The second table is called the subservient table.

Consider two hypothetical tables, **employees** and **depts**, which contain the following columns and rows (dash “—” indicates a NULL value):

**employees**

emp_num	dept_num
2	105
4	103
6	103
5	—
3	102

**depts**

dept_num	dept_loc
102	NY
103	LA
105	SF

Suppose, for example, that you need a list of employee numbers and department locations for all employees, including those employees whose department locations are unknown (represented by NULL values in the **employees** table). The following query fetches the desired results:

---

```
SELECT emp_num, dept_loc
FROM employees, OUTER depts
WHERE employees.dept_num = depts.dept_num
```

---

The keyword **OUTER** designates **depts** as the subservient table, making **employees** the dominant table. **INFORMIX-ESQL/C** processes the query as follows:

1. **INFORMIX-ESQL/C** applies filters to the subservient table while sequentially applying the join condition to the rows of the dominant table. Rows in the dominant table are retrieved without considering the join, but rows from the subservient table (outer table) are retrieved only if they satisfy the join condition. Any dominant-table rows that do not have a matching row from the subservient table receive a row of nulls in place of a subservient-table row.

The result is a "table" with the following rows:

emp_num	dept_num	dept_num	dept_loc
2	105	105	SF
3	102	102	NY
4	103	103	LA
5	—	—	—
6	103	103	LA

**Note:** A *filter* is a condition expressed in a WHERE clause that applies to columns in a single table, for example,

"dept\_loc = SF"

or

"emp\_num < 105"

Because **INFORMIX-ESQL/C** applies such filters to the subservient table as it performs the join, the resulting "table" may contain NULL values that were not present in the subservient table prior to the join.

Suppose the sample query includes a filter on the **dept\_loc** column:

---

```
SELECT emp_num, dept_loc
FROM employees, OUTER depts
WHERE employees.dept_num = depts.dept_num
AND dept_loc != "LA"
```

---

At Step 2, the results include more rows of NULL values than the results of the original query:

emp_num	dept_num	dept_num	dept_loc
2	105	105	SF
3	102	102	NY
4	—	—	—
5	—	—	—
6	—	—	—

The filter removes rows from the **depts** table where **dept\_loc** is equal to LA.

2. After performing the join, **INFORMIX-ESQL/C** applies filters to the dominant table (if they exist).
3. **INFORMIX-ESQL/C** applies the **SELECT** clause to eliminate unneeded columns, and the query returns the results.

emp_num	dept_loc
2	SF
3	NY
4	LA
5	—
6	LA

In a similar way to the previous example, the following query produces a list of all customers with supplemental information for those customers with items on orders. Where **orders.customer\_num** is not equal to **customer.customer\_num**, **INFORMIX-ESQL/C** combines a row of NULL values with the corresponding row from the **customer** table. Because the query does not contain filters, the results preserve every row from the dominant table.



### Query 3

---

```
SELECT customer.customer_num, company, order_num, ship_date
FROM customer, OUTER orders
WHERE customer.customer_num = orders.customer_num
```

---

### Query 3 Results

customer_num	company	order_num	ship_date
101	All Sports Supplies	1002	06/06/1984
102	Sports Spot		
103	Phil's Sports		
104	Play Ball!	1001	06/05/1984
104	Play Ball!	1003	06/07/1984
104	Play Ball!	1011	06/07/1984
104	Play Ball!	1013	06/11/1984
105	Los Altos Sports		
106	Watson & Son	1004	
106	Watson & Son	1014	06/09/1984
107	Athletic Supplies		
108	Quinn's Sports		
109	Sport Stuff		
110	AA Athletics	1008	06/27/1984
110	AA Athletics	1015	06/11/1984
111	Sports Center	1009	06/15/1984
112	Runners & Others	1006	
113	Sportstown		
114	Sporting Place		
115	Gold Medal Sports	1010	06/07/1984
116	Olympic City	1005	06/08/1984
117	Kids Korner	1007	06/08/1984
117	Kids Korner	1012	06/09/1984
118	Blue Ribbon Sports		

The preceding example queries two tables in the simplest form of an outer join. You can, in fact, use outer joins to query any number of tables, producing more forms than can be discussed here. The following forms are possible when three tables are involved in a query:

- You can outer-join the result of a simple join to a third table.

---

```
SELECT column-list
FROM x, OUTER (y,z)
WHERE x.a = y.a AND y.b = z.b
```

---

Query 4 performs this kind of join (see the following section "Examples").

- You can outer-join the result of an outer join to a third table.

---

```
SELECT column-list
FROM x, OUTER (y, OUTER z)
WHERE x.a = y.a AND y.b = z.b
```

---

*or*

---

```
SELECT column-list
FROM x, OUTER (y, OUTER z)
WHERE x.a = z.a AND y.b = z.b
```

---

Queries 5 and 6 perform this kind of join (see the following section “Examples”).

- You can outer-join two tables individually to a third table, in which case, join relationships are possible only between the subservient tables and the dominant table. Query 7 performs this kind of join (see the following section “Examples”).

---

```
SELECT column-list
FROM x, OUTER y, OUTER z
WHERE x.a = y.a AND x.b = z.b
```

---

When you outer-join several tables to another table, make sure your WHERE clause does not specify impossible join conditions. The following query attempts a join between two subservient tables:

---

```
SELECT column-list
FROM x, OUTER y, OUTER z
WHERE x.a = y.a AND y.b = z.b
```

---

An INFORMIX-ESQL/C error results; every outer join must have a dominant table.

The following examples use the **stores** database to demonstrate common multiple-table outer joins.

## Examples

### Query 4

This query outer-joins the result of a simple join to a third table. It produces a list of all customers with supplemental information (order number, stock number, manufacturer code, and quantity ordered) for those customers who have ordered items manufactured by Anza.

---

```
SELECT customer.customer_num, lname,  
       orders.order_num, stock_num, manu_code, quantity  
FROM customer, OUTER (orders, items)  
WHERE customer.customer_num = orders.customer_num AND  
       orders.order_num = items.order_num AND  
       manu_code = "ANZ"
```

---

INFORMIX-ESQL/C performs the simple join between **orders** and **items** first, yielding information on all orders for Anza-manufactured items. The outer join combines the **customer** table with the Anza order information. The query results do not include orders for other items.

### Query 4 Results

customer_num	lname	order_num	stock_num	manu_code	quantity
101	Pauli				
102	Sadler				
103	Currie				
104	Higgins	1003	9	ANZ	1
104	Higgins	1003	8	ANZ	1
104	Higgins	1003	5	ANZ	5
104	Higgins	1011	5	ANZ	5
104	Higgins	1013	5	ANZ	1
104	Higgins	1013	6	ANZ	1
104	Higgins	1013	9	ANZ	2
105	Vector				
106	Watson				
107	Ream				
108	Quinn				
109	Miller				
110	Jaeger	1008	8	ANZ	1
110	Jaeger	1008	9	ANZ	5
111	Keyes				
112	Lawson	1006	5	ANZ	5
112	Lawson	1006	6	ANZ	1
113	Beatty				
114	Albertson				
115	Grant	1010	6	ANZ	1
116	Parmelee	1005	5	ANZ	10
116	Parmelee	1005	6	ANZ	1
117	Sipes	1012	8	ANZ	1
117	Sipes	1012	9	ANZ	10
118	Baxter				

## Query 5

This query outer-joins the result of an outer join to a third table. When you use a nested outer join, the query preserves order numbers that Query 4 (using a nested simple join) eliminates. The query results include all orders, whether or not they contain Anza-manufactured items. For other items, the condition

---

```
where manu_code = "ANZ"
```

---

eliminates stock numbers, manufacturer codes, and quantities as before.

---

```
SELECT customer.customer_num, lname,  
       orders.order_num, stock_num, manu_code, quantity  
FROM customer, OUTER (orders, OUTER items)  
WHERE customer.customer_num = orders.customer_num AND  
       orders.order_num = items.order_num AND  
       manu_code = "ANZ"
```

---



Query 5 Results

customer_num	lname	order_num	stock_num	manu_code	quantity
101	Pauli	1002			
102	Sadler				
103	Currie				
104	Higgins	1001			
104	Higgins	1003	9	ANZ	1
104	Higgins	1003	8	ANZ	1
104	Higgins	1003	5	ANZ	5
104	Higgins	1011	5	ANZ	5
104	Higgins	1013	5	ANZ	1
104	Higgins	1013	6	ANZ	1
104	Higgins	1013	9	ANZ	2
105	Vector				
106	Watson	1004			
106	Watson	1014			
107	Ream				
108	Quinn				
109	Miller				
110	Jaeger	1008	8	ANZ	1
110	Jaeger	1008	9	ANZ	5
110	Jaeger	1015			
111	Keyes	1009			
112	Lawson	1006	5	ANZ	5
112	Lawson	1006	6	ANZ	1
113	Beatty				
114	Albertson				
115	Grant	1010	6	ANZ	1
116	Parmelee	1005	5	ANZ	10
116	Parmelee	1005	6	ANZ	1
117	Sipes	1007			
117	Sipes	1012	8	ANZ	1
117	Sipes	1012	9	ANZ	10
118	Baxter				

In addition to customer, orders, and so on, the following queries include a hypothetical table named custnotes, containing the following columns and data:

customer_num	notes
104	sponsors soccer team
108	customer for 20 years
115	opening a second store
118	new customer

## Query 6

This query produces a list of all customers with order numbers and selected notes.

---

```
SELECT customer.customer_num, orders.order_num, notes
FROM customer, OUTER (orders, OUTER custnotes)
WHERE customer.customer_num = orders.customer_num AND
orders.customer_num = custnotes.customer_num
```

---

The outer join between **custnotes** and **orders** preserves notes only for customers who also have orders.

### Query 6 Results

customer_num	order_num	notes
101	1002	
102		
103		
104	1001	sponsors soccer team
104	1003	sponsors soccer team
104	1011	sponsors soccer team
104	1013	sponsors soccer team
105		
106	1004	
106	1014	
107		
108		
109		
110	1008	
110	1015	
111	1009	
112	1006	
113		
114		
115	1010	opening a second store
116	1005	
117	1007	
117	1012	
118		

To preserve notes for customers 108 and 118 who do not have orders, you must outer-join the **custnotes** table *directly* with the **customer** table, as shown in the next query.

# Query 7

This query outer-joins two tables individually to a third table. It outer-joins both **orders** and **custnotes** to **customer** (the dominant table).

```
SELECT customer.customer_num, orders.order_num, notes
FROM customer, OUTER orders, OUTER custnotes
WHERE customer.customer_num = orders.customer_num AND
customer.customer_num = custnotes.customer_num
```

Customer notes now appear whether or not customers have orders.

## Query 7 Results

customer_num	order_num	notes
101	1002	
102		
103		
104	1001	sponsors soccer team
104	1003	sponsors soccer team
104	1011	sponsors soccer team
104	1013	sponsors soccer team
105		
106	1004	
106	1014	
107		
108		customer for 20 years
109		
110	1008	
110	1015	opening a second store
111	1009	
112	1006	
113		
114		
115	1010	
116	1005	
117	1007	
117	1012	
118		new customer

All of the preceding queries fetch information from one table with supplemental information from other tables. When you need similar results, Informix recommends you use an outer join. When you do not need supplemental information, as is normally the case, use a simple join instead. Be aware that your choice of an outer join can influence query optimization and processing.

# **Appendix H**

## **Working with DATETIME and INTERVAL Data**





## Overview

The DATETIME and INTERVAL data types provide a way of storing moments in time and the spans between moments. The DATETIME data type holds a value that represents a single point in time. You choose how precisely a DATETIME value is stored; its precision can range from a year through a fraction of a second. The INTERVAL data type holds a value that represents a span of time; it can represent either a span of years and months or a span of days, hours, minutes, seconds, and fractions of a second.

You enter DATETIME and INTERVAL values in a form that explicitly identifies not only the value but also the data type (DATETIME or INTERVAL) and the precision, or you enter values as character strings that include only the values. The explicit form, sometimes called a *literal*, appears throughout this appendix (except as noted) because you can use this form in more situations than a character string.

You can manipulate DATETIME and INTERVAL values in a number of ways. You can enter them as data, you can use them in relational expressions, and you can manipulate them arithmetically. This appendix supplies you with guidelines on how you can use DATETIME and INTERVAL values. Note, however, that many of the examples presented here are not in context; that is, they do not include the full syntax necessary for an executable SQL or programming statement. (The section “Data Manipulation Statements” presents examples of complete SQL data manipulation statements that use DATETIME and INTERVAL values.)

## DATETIME Columns

The DATETIME data type is composed of a contiguous sequence of fields that represent each component of time you want to record. Here is the syntax for defining a DATETIME column:

---

*column-name* DATETIME *first* TO *last*

---

where *first* and *last* are fields taken from the following list:

Field	Valid Entries
YEAR	A year numbered from 1 to 9999.
MONTH	A month numbered from 1 to 12.
DAY	A day numbered from 1 to 31, as appropriate to the month in question.
HOUR	An hour numbered from 0 (midnight) to 23.
MINUTE	A minute numbered from 0 to 59.
SECOND	A second numbered from 0 to 59.
FRACTION	A decimal fraction of a second with up to five digits of precision. The default precision is three digits (thousandth of a second). Other precisions are indicated explicitly by writing FRACTION( <i>n</i> ), where <i>n</i> is the desired number of digits from 1 to 5.

A DATETIME column need not include all fields from YEAR to FRACTION; it can be a subset of fields or even a single field. You can define a DATETIME column to include only those fields you need. For example, you may define a column as MONTH TO HOUR. However, the fields you include must be contiguous; that is, they must include all fields in sequence. For example, you cannot include just MONTH and HOUR. You must also include DAY as part of that value.

## Entering DATETIME Values

There is a rigid syntax that you must observe when you enter a DATETIME constant into a DATETIME column. If you use the literal form, you must supply the actual date and time values, with proper delimiters between the fields, and a list of the first and last fields you are specifying. (A later section of this appendix describes character string formats.)

Just as you can define a DATETIME column to be only a subset of all possible fields, you can also enter values that contain just a subset of the defined fields from a DATETIME column. For example, you can enter a value of MONTH TO HOUR into a column that is defined as YEAR TO MINUTE. However, each entered value must always



contain information for a contiguous sequence of fields. For example, you cannot enter a value for just MONTH and HOUR; the entry must include a value for DAY as well.

When you enter a value with fewer fields than the defined column, the value you enter is automatically expanded to fill all the defined fields. If you leave out a more significant field, that is, a field of larger magnitude than any value you supply, that field(s) is automatically filled with the current date. In the previous example, YEAR is left out, so the current year is automatically inserted into your entry. Alternatively, if you leave out a less significant field, that field is filled with zeros (or one for MONTH and DAY) in your entry. In the previous example, MINUTE is left out, so zeros are inserted into the MINUTE field for your entry.

Consider the following example:

---

```
CREATE TABLE mytable (mytime DATETIME YEAR TO MINUTE)
INSERT INTO mytable VALUES (DATETIME (8-31 12) MONTH TO HOUR)
```

---

Even though the column **mytime** has a defined scope of YEAR TO MINUTE, you can supply information for only MONTH TO HOUR. The entered value is automatically expanded to fill the column. If the current year is 1989, the actual inserted value becomes:

---

```
DATETIME (1989-8-31 12:00) YEAR TO MINUTE
```

---

Notice that the current year, in this case 1989, has been added automatically along with zeros for the MINUTE field.

The current date is also used to evaluate whether a DATETIME value that does not include precision up to YEAR is an acceptable date. This can cause problems when the largest precision of the value is DAY. For example, assume the current month is September, and you attempt to execute the following statements:



---

```
CREATE TABLE mytable (mytime DATETIME DAY TO MINUTE)
INSERT INTO mytable VALUES (DATETIME (31 12) DAY TO HOUR)
```

---

Before the row is inserted, the validity of the DATETIME value is checked against the current date. Because September has only 30 days, the 31 for DAY is out of range and will produce an error. As a result, you cannot insert this row during a month that has fewer than 31 days.

---

**Caution!** In order to eliminate the risk of processing difficulties during months with fewer than 31 days, do not create DATETIME columns with a *first* precision of DAY, and do not enter DATETIME values with DAY as the field of largest precision.

---

When you enter a DATETIME value, you must include a *qualifier* that specifies both the first (largest) and last (smallest) field in the value you are about to enter. Including the qualifier is necessary because, as noted earlier, the value you are entering may contain fewer fields than defined for that column. Acceptable qualifiers for the first and last fields are identical to the list of valid DATETIME fields displayed previously.

A valid entry contains the DATETIME name, the values to be entered, and the field qualifier. Use the syntax *first-field TO last-field* to qualify the DATETIME value. For example, in the following entry, YEAR TO FRACTION indicates that values for all possible fields are included in the values listed between the parentheses. The order of entry and the type of delimiter identify which value corresponds to which date component.

---

```
DATETIME (1985-5-2 14:05:00.000) YEAR TO FRACTION
```

---

## **List of Appendixes**

**Appendix A. Header Files**

**Appendix B. System Catalogs**

**Appendix C. Environment Variables**

**Appendix D. Reserved Words**

**Appendix E. The *stores* Database**

**Appendix F. INFORMIX-ESQL/C  
Utility Programs**

**Appendix G. Outer Joins**

**Appendix H. Working with DATETIME  
and INTERVAL Data**



The following examples illustrate values that do not include all fields:

---

DATETIME (89-8-16) YEAR TO DAY  
DATETIME (16 12:23) DAY TO MINUTE  
DATETIME (21.234) SECOND TO FRACTION  
DATETIME (32) SECOND TO SECOND  
DATETIME (.0032) FRACTION TO FRACTION(4)

---

When YEAR is given as a two-digit number, as in the first example, it is automatically changed to "19xx" ("1989" in this case). If you want to specify years 1-99, pad the entry with a preceding zero(s), for example "089" or "0089." When a value contains just one field, the first and last qualifiers are the same. If the first and last qualifiers are both FRACTIONS, you specify the precision only for the last field.

Values for the fields are written as integers and are separated by delimiters. All field values are two-digit integers, except for the YEAR (four digits) and FRACTION (*n* digits) fields. The following delimiters are used with DATETIME values:

<b>Delimiter</b>	<b>Placement in DATETIME Expression</b>
hyphen	Between the YEAR and MONTH, and the MONTH and DAY portions of the value.
space	Between the DAY and HOUR portions of the value.
colon	Between the HOUR and MINUTE, and the MINUTE and SECOND portions of the value.
decimal point	Between the SECOND and FRACTION portions of the value.



# INTERVAL Columns

Like the DATETIME data type, the INTERVAL data type is composed of a contiguous sequence of fields. Here is the syntax for defining an INTERVAL column:

---

*column-name* INTERVAL *first* TO *last*

---

where *first* and *last* are fields taken from *one* of the following two lists:

Field	Valid Entries
-------	---------------

YEAR	A number of years.
------	--------------------

MONTH	A number of months.
-------	---------------------

OR

DAY	A number of days.
-----	-------------------

HOURL	A number of hours.
-------	--------------------

MINUTE	A number of minutes.
--------	----------------------

SECOND	A number of seconds.
--------	----------------------

FRACTION	A decimal fraction of a second, with up to five digits of precision. The default precision is three digits (thousandth of a second). Other precisions are indicated explicitly by writing FRACTION( <i>n</i> ), where <i>n</i> is the desired number of digits from 1 to 5.
----------	---

Like a DATETIME column, you can define an INTERVAL column to include only the fields you need. An INTERVAL column can contain either YEAR and MONTH information or DAY through FRACTION information, but not a combination of both. The reason for this is that the INTERVAL data type is designed specifically to represent a span of time independent of actual dates. The number of days in a month depends on which month it is. INTERVAL data cannot be month dependent; therefore, it is not possible to combine months and days in an INTERVAL value.

## Entering INTERVAL Values

Like its DATETIME counterpart, a value you enter into an INTERVAL column need not include all fields contained in the column. For example, you can enter a value of HOUR TO SECOND into a column defined as DAY TO SECOND. However, a value must always consist of a contiguous sequence of fields. For example, you cannot enter just HOURS and SECONDS; you must include MINUTES as well.

When you enter a value in an INTERVAL column, you must include a *qualifier* that specifies both the first (largest) and last (smallest) fields in the value, just as you did for DATETIME values. In addition, you can optionally specify the precision of the first field (and the last field if it is a FRACTION). Acceptable qualifiers for the first field and last field are identical to the list of INTERVAL fields previously displayed.

You write INTERVAL values in the same literal format as DATETIME values; a valid entry contains the INTERVAL name, the values to be entered, and the field qualifier. For example, the following entries are valid INTERVAL values:

---

```
INTERVAL (5-3) YEAR TO MONTH
INTERVAL (9) MONTH TO MONTH
INTERVAL (12:23) HOUR TO MINUTE
```

---

Each entry represents a span of time: 5 years and 3 months, 9 months, and 12 hours and 23 minutes, respectively. None of the entries refer to a specific point in time. For example, the nine-month interval could represent a span from any single year or across any two years.

When a value contains just one field, the first and last qualifiers are the same. If the first and last qualifiers are both FRACTIONS, you can only specify the precision in the last field.

The first field in an INTERVAL value can be up to nine digits in size (except for FRACTIONS which cannot be more than five digits), but if the value you wish to enter is greater than the default number of digits allowed for that field, you must explicitly identify the number of significant digits in the value you are entering. By default you are allowed up to four digits in a year, three digits in a fraction, and two digits in all other fields. If you need more digits than allowed, you enter the number of required digits in the field qualifier.

---

```
INTERVAL (10000-2) YEAR(5) TO MONTH
INTERVAL (127) MONTH(3) TO MONTH
INTERVAL (365 0:23) DAY(3) TO MINUTE
INTERVAL (1421 11:36:54.93721) DAY(4) TO FRACTION(5)
```

---

The first example illustrates how you can specify a span of 10,000 years by including the number five next to the YEAR. The second example allows a three-digit MONTH entry. Notice that the number is attached to the first qualifier. You will get a syntax error if you attach the number to the last qualifier. The third example is a DAY TO MINUTE span of 1 year and 23 minutes. This illustrates how you can have values of any length by converting years and months into their corresponding number of days. Note also how you can include a zero for the HOUR field in order to enter a value just for MINUTE. The last example shows how you can specify the precision at both ends when FRACTION is the last entered field.

INTERVAL data follow normal date and time conventions concerning the range of acceptable values you can enter. You get an error if you exceed the possible values for a given field. Note these examples:

---

```
INTERVAL (5-13) YEAR TO MONTH
INTERVAL (6-1) YEAR TO MONTH
```

---

The first example causes an error because 13 months is more than a year. The second example correctly expresses the same span of time.

The values for the fields are written as integers and are separated by delimiters. The following delimiters are used with INTERVAL values:

Delimiter	Placement in INTERVAL Expression
hyphen	Between the YEAR and MONTH portions of the value.
space	Between the DAY and HOUR portions of the value.
colon	Between the HOUR, MINUTE, and SECOND portions of the value.
decimal point	Between the SECOND and FRACTION portions of the value.



## ***DATETIME and INTERVAL Values as Character Strings***

You can enter DATETIME and INTERVAL data in the literal forms described in the previous sections or as quoted character strings. The following examples illustrate both formats:

---

```
CREATE TABLE mytable(mytime DATETIME YEAR TO DAY, myval INTERVAL DAY TO SECOND)

INSERT INTO mytable(mytime) VALUES (DATETIME(89-5-12) YEAR TO DAY)
INSERT INTO mytable(mytime) VALUES ("89-5-12")

INSERT INTO mytable(myval) VALUES (INTERVAL(17 11:15:30) DAY TO SECOND)
INSERT INTO mytable(myval) VALUES ("17 11:15:30")
```

---

Values that are entered as character strings are automatically converted into DATETIME or INTERVAL values. You may use character strings whenever you enter information for all fields defined for that DATETIME or INTERVAL column.

When a character string is converted into a DATETIME or INTERVAL value, the engine assumes that the character string includes information about all the defined fields. You cannot use character strings to enter DATETIME or INTERVAL values for a subset of fields, because this produces ambiguous values. If the character string does not contain information for all fields, the engine returns an error. For example:

---

```
INSERT INTO mytable(mytime) VALUES (DATETIME(5-12) MONTH TO DAY)
INSERT INTO mytable(mytime) VALUES ("5-12")

INSERT INTO mytable(myval) VALUES (INTERVAL(11:15) HOUR TO MINUTE)
INSERT INTO mytable(myval) VALUES ("11:15")
```

---

The previous DATETIME example (column **mytime**) enters a MONTH and DAY value into a column defined as YEAR TO DAY. Entering only these values is appropriate in the first INSERT statement because you tell the engine there is no year information. In this case, the engine automatically adds the current year. However, the character string does not indicate what information is left out. The engine does not know whether the "5-12" refers to YEAR TO MONTH or MONTH TO DAY. It, therefore, returns an error. The previous INTERVAL example (column **myval**) enters an HOUR and MINUTE value into a column defined as DAY TO SECOND. The first INSERT statement simply pads the value with zeros for YEAR and SECOND. The second



INSERT statement produces a conversion error because the engine does not know whether the “11:15” refers to HOUR TO MINUTE or MINUTE TO SECOND.

## ***Range of Operations Using DATETIME and INTERVAL***

You can use DATETIME and INTERVAL data in a variety of arithmetic and relational expressions. You can manipulate a DATETIME value in conjunction with another DATETIME value, an INTERVAL value, the current time (identified by the keyword CURRENT), or an unspecific unit of time (identified by the keyword UNITS). You can manipulate an INTERVAL value in conjunction with the same choices as a DATETIME value. In addition, you can multiply or divide an INTERVAL value by a number.

An INTERVAL column can hold a value that represents the difference between two DATETIME values or the difference (or sum) between two INTERVAL values. In either case, the result is a span of time, which is an INTERVAL value. On the other hand, if you add or subtract an INTERVAL value from a DATETIME value, you get another DATETIME value because the result is a specific point in time.

The following table indicates the range of expressions that you can use with DATETIME and INTERVAL data, along with the data type resulting from each expression.

Data Type of Operand 1	Operator	Data Type of Operand 2	Result
Datetime	-	Datetime	Interval
Datetime	+ or -	Interval	Datetime
Interval	+	Datetime	Datetime
Interval	+ or -	Interval	Interval
Datetime	-	CURRENT	Interval
CURRENT	-	Datetime	Interval
Interval	+	CURRENT	Datetime
CURRENT	+ or -	Interval	Datetime
Datetime	+ or -	UNITS	Datetime
Interval	+ or -	UNITS	Interval
Interval	* or /	Number	Interval

No other combinations are allowed. (You can substitute DATE values for DATETIME values in most situations. See the last section in this appendix for more information.) You cannot add two DATETIME values because this operation does not produce either a point in time or a span of time. For example, you cannot add December 25 to January 1, though you can subtract it to find the span between the two dates. Examples for each expression are presented in the following sections.

## Manipulating DATETIME Values

You can subtract most DATETIME values from each other. Dates can be in any order; the result will be either a positive or a negative INTERVAL. The first DATETIME value determines the field precision for the result. If the second DATETIME value has fewer fields than the first, the shorter value is automatically extended to match the longer one. If the second DATETIME value has more fields than the first (regardless of whether the precision of the extra fields is larger or smaller than those in the first value), the additional fields in the second value are ignored in the calculation.

---

```
DATETIME (1989-9-30 12:30) YEAR TO MINUTE
- DATETIME (1989-8-1 11) YEAR TO HOUR
result: INTERVAL (60 01:30) DAY TO MINUTE
```

```
DATETIME (1989-9-30) YEAR TO DAY
- DATETIME (10-1) MONTH TO DAY
result: INTERVAL (-1) DAY TO DAY [assuming current year is 1989]
```

---

In the first example, minutes are not included for the second value; therefore, the minutes are automatically set to zero, and the resulting INTERVAL is 60 days, 1 hour, and 30 minutes. In the second example, the year is not included for the second value; therefore, the year is automatically set to the current year, in this case 1989, and the resulting INTERVAL is negative reflecting that the second date is later than the first.

## Manipulating DATETIME with INTERVAL Values

INTERVAL values can be added to or subtracted from DATETIME values. In either case, the result is a DATETIME value. If you are adding an INTERVAL to a DATETIME, the order of values is unimportant; however, if you are subtracting, the DATETIME value must come first.

---

```
DATETIME (1989-8-1) YEAR TO DAY
+ INTERVAL (3-5) YEAR TO MONTH
result: DATETIME (1993-01-01) YEAR TO DAY
```

```
DATETIME (15:45) HOUR TO MINUTE
- INTERVAL (720) MINUTE(3) TO MINUTE
result: DATETIME (3:45) HOUR TO MINUTE
```

---

Adding or subtracting an INTERVAL value simply moves the DATETIME value forward or backward. The first example moves the date ahead three years and five months. The second example moves the value back 12 hours. However, this amount is presented as 720 minutes, a three-digit number. This precision exceeds the default of two digits and requires that you explicitly define the first qualifier as MINUTES(3).

In most situations the database engine automatically adjusts the calculation when the initial values do not have the same precision. However, there are certain situations where you must explicitly adjust the precision of one value in order to perform the calculation. One



such situation is where the INTERVAL value you are adding or subtracting has fields not included in the DATETIME value. The DATETIME value is not automatically extended to match the INTERVAL value; you must do that explicitly through the EXTEND() function. (See the EXTEND() syntax explanation for more information.) For example, you cannot directly subtract the 720 minute INTERVAL in the second example from the DATETIME value in the first example that has a precision from YEAR TO DAY. You can perform this calculating by using the EXTEND() function.

---

```
EXTEND (DATETIME (1989-8-1) YEAR TO DAY, YEAR TO MINUTE)
- INTERVAL (720) MINUTE(3) TO MINUTE
result:  DATETIME (1989-07-31 12:00) YEAR TO MINUTE
```

---

The EXTEND() function explicitly increases the DATETIME precision from YEAR TO DAY to YEAR TO MINUTE. This allows the database engine to perform the calculation, and the result has the EXTENDED precision of YEAR TO MINUTE.

## Manipulating INTERVAL Values

INTERVAL values can be added or subtracted from each other as long as both values are of the same type; that is, both are YEAR TO MONTH or both are DAY TO FRACTION.

---

```
INTERVAL (5-3) YEAR TO MONTH
+ INTERVAL (15) MONTH TO MONTH
result:  INTERVAL (6-06) YEAR TO MONTH

INTERVAL (100:30.0005) MINUTE(3) TO FRACTION(4)
- INTERVAL (120.01) SECOND(3) TO FRACTION
result:  INTERVAL (98:29.9905) MINUTE TO FRACTION(4)
```

---

In the first example, a MONTH TO MONTH value is added to a YEAR TO MONTH value. The 15 months is automatically converted into years and months when it is added, resulting in a total INTERVAL value of 6 years and 6 months. In the second example, a SECOND TO FRACTION value is subtracted from a MINUTE TO FRACTION value. Note the use of numeric qualifiers to alert the engine that the MINUTE and FRACTION in the first value and the SECOND in the second value exceed the default number of digits.



When adding or subtracting INTERVAL values, the second value cannot have a field with greater precision than the first. For example, the second INTERVAL cannot be YEAR TO MONTH if the first is MONTH TO MONTH. The second INTERVAL can have a field of smaller precision than the first. For example, the second INTERVAL can be HOUR TO SECOND when the first is DAY TO HOUR. The additional fields (in this case MINUTE and SECOND) in the second INTERVAL value are ignored in the calculation.

## Operations Using CURRENT Values

CURRENT is a keyword that evaluates to the current date and time. (See the CURRENT syntax explanation for more information.) You can use it in place of a specific DATETIME value. The following examples repeat previous examples, except CURRENT replaces one of the DATETIME values.

---

```
example current date:  DATETIME (1989-8-1 00:00:00.000) YEAR TO FRACTION

DATETIME (1989-9-30 12:30) YEAR TO MINUTE - CURRENT
result:  INTERVAL (60 12:30) DAY TO MINUTE

CURRENT - DATETIME (10-1) MONTH TO DAY
result:  INTERVAL (-61 00:00:00.000) DAY TO FRACTION

CURRENT + INTERVAL (3-5) YEAR TO MONTH
result:  DATETIME (1993-01-01 00:00:00.000) YEAR TO FRACTION

CURRENT - INTERVAL (720) MINUTE(3) TO MINUTE
result:  DATETIME (1989-07-31 12:00:00.000) YEAR TO FRACTION
```

---

The precision of the first value determines the precision of output when using DATETIME values. In the first example, the precision used for CURRENT matches the first DATETIME value, YEAR TO MINUTE; the resulting INTERVAL value has precision DAY TO MINUTE. The precision of the other three examples is determined by CURRENT, which uses the default precision of YEAR TO FRACTION.

## Operations Using UNITS Values

A shorthand way to represent single-field INTERVAL values is to use the UNITS keyword. It allows you to quickly specify amounts of a field--for example, number of years--that you want to add to or subtract from a DATETIME or INTERVAL value. UNITS has the following syntax:

---

*number* UNITS *field-name*

---

where *number* is the amount and *field-name* is the type of field. The following examples repeat previous examples, except UNITS replaces one of the INTERVAL values.

---

DATETIME (1989-8-1) YEAR TO DAY + 3 UNITS YEAR

result: DATETIME (1992-08-01) YEAR TO DAY

DATETIME (15:45) HOUR TO MINUTE - 720 UNITS MINUTE

result: DATETIME (3:45) HOUR TO MINUTE

INTERVAL (5-3) YEAR TO MONTH + 15 UNITS MONTH

result: INTERVAL (6-06) YEAR TO MONTH

INTERVAL (100:30.0005) MINUTE(3) TO FRACTION(4) - 120 UNITS SECOND

result: INTERVAL (98:30.0005) MINUTE TO FRACTION(4)

---

In each of these cases, an amount of a single field is added to or subtracted from an INTERVAL or DATETIME value. Note that the second and third examples produce the same results as before. This is because manipulating single-field INTERVAL values is identical to manipulating UNITS values of the same type and amount. The UNITS keyword is simply an easy way to write such INTERVAL values.

## Multiplying or Dividing INTERVAL Values

You can multiply or divide INTERVAL values by a number. The number can be an integer or a fraction. However, INTERVAL values cannot be expressed as a fraction of a field. All results are written only as integers. If there is a remainder from the calculation, it is ignored, and the result is truncated. Note the following examples:

---

INTERVAL (13-5) YEAR TO MONTH / 2

result: INTERVAL (6-8) YEAR TO MONTH

INTERVAL (15:30.0002) MINUTE TO FRACTION(4) \* 2.5

result: INTERVAL (38:45.0005) MINUTE TO FRACTION(4)

---

The results of any calculation include the same amount of precision as the original INTERVAL value. In the first example, the YEAR value is divided by two leaving one year as a remainder. This remainder is converted into 12 months, added to the 5 existing months, and then

divided. One month is left over, but there is no lower precision, so this remainder is simply discarded in the final result of 6 years and 8 months.

The second example multiplies an INTERVAL by a fraction. In this case,  $15 * 2.5 = 37.5$  minutes,  $30 * 2.5 = 75$  seconds, and  $2 * 2.5 = 5$  fraction(4). The .5 minute is converted into 30 seconds and 60 seconds are converted into 1 minute, which produces the final result of 38 minutes, 45 seconds, and .0005 of a second.

## *Data Manipulation Statements*

You can use DATETIME and INTERVAL values in any place that is appropriate for other data types. Frequently, you might use these values in the context of an SQL data manipulation statement; namely, INSERT, UPDATE, DELETE, or SELECT.

In a previous section, you saw how you can use DATETIME and INTERVAL values in an INSERT statement. The following examples illustrate how you can use these values in other types of SQL data manipulation statements.

---

```
CREATE TABLE mytable(mytime DATETIME YEAR TO DAY, myval INTERVAL DAY TO SECOND)

UPDATE mytable
  SET myval = myval + INTERVAL (1 1:1:1) DAY TO SECOND

DELETE FROM mytable WHERE mytime < CURRENT - 1 UNITS YEAR

SELECT * FROM mytable WHERE mytime BETWEEN
  DATETIME(88-7-1) YEAR TO DAY and DATETIME (89-6-30) YEAR TO DAY

SELECT mytime, mytime + INTERVAL (45) DAY TO DAY from mytable
```

---

The first example updates all rows by a constant INTERVAL value. The second example deletes the rows that are more than a year old. The third example selects all rows for a given one-year period. The final example includes a derived field in the select-list so that you can compare the existing date with a later one.



## ***DATETIME and DATE Compatibility***

The database engine attempts to make appropriate data type conversions when they are required. For most situations, this allows you to use a DATETIME value wherever it is appropriate to use a DATE value and vice versa.

You can use DATE values in arithmetic expressions with DATETIME or INTERVAL values. You can write expressions that allow the following manipulations:

---

DATE - DATETIME	result is INTERVAL
DATETIME - DATE	result is INTERVAL
DATE +/- INTERVAL	result is DATETIME

---

In these cases, DATE values are first converted to their corresponding DATETIME equivalents, and then the expression is computed in the normal way. You can also use TODAY, CURRENT, and UNITS to represent DATE, DATETIME, and INTERVAL values, respectively, in such expressions.

While you can interchange DATE and DATETIME values in many situations, it must be clear whether a value is a DATE or a DATETIME. A DATE value can come from any of the following sources:

- a column or program variable of type DATE
- the TODAY keyword
- the DATE() function

A DATETIME value can come from any of the following sources:

- a column or program variable of type DATETIME
- the CURRENT keyword
- the EXTEND() function
- a DATETIME literal

You can also represent DATE and DATETIME values as quoted character strings. However, you can only use strings that are in proper DATE or DATETIME format and only in contexts where the corresponding data type of the string is known. The following examples illustrate what string format is required in various contexts:



---

```
DATE ("date-string")
WHERE TODAY > "date-string"
WHERE DATE-column < "date-string"

EXTEND ("datetime-string" [,qualifier])
WHERE DATETIME-column > "datetime-string"
```

---

When a DATE value is expected, the string must be in DATE format; when a DATETIME value is expected, the string must be in DATETIME format. For example, you can use the string "10/30/1989" as a *date-string* but not as a *datetime-string*. Instead, you must use "1989-10-30" or "89-10-30" as the *datetime-string*.

**Note:** You can also subtract a DATE value from another DATE value, but the result is a positive or negative INTEGER value, rather than an INTERVAL value. If an INTERVAL value is required, you can either convert the INTEGER into an INTERVAL or convert one of the DATE values into a DATETIME before subtracting.

For example, the following expression uses the DATE() function to convert character string constants to DATE values, calculates their difference, and then uses the UNITS DAY keywords to convert the INTEGER result to an INTERVAL value:

---

```
(DATE ("5/2/1989") - DATE ("4/6/1954")) UNITS DAY
result:  INTERVAL (12810) DAY(5) TO DAY
```

---

If you need YEAR TO MONTH precision, you can use the EXTEND() function on the first DATE operand, as illustrated in the next example:

---

```
EXTEND (DATE ("5/2/1989"), YEAR TO MONTH) - DATE ("4/6/1954")
result:  INTERVAL (35-01) YEAR TO MONTH
```

---

Note that the resulting INTERVAL precision is YEAR TO MONTH because the DATETIME value came first. If the DATE value had come first, the resulting INTERVAL precision would have been DAY(5) TO DAY.

# **Glossary**



***abort.*** To interrupt an active process before completion. The status of data may or may not be the same as it was before the process began.

***access method.*** A procedure used to retrieve rows from or insert rows into a storage location. In the SET EXPLAIN statement, access method refers to the type of table access in a query, for example, SEQUENTIAL SCAN versus INDEX PATH.

***access mode.*** The status of an open file determines read and write access to that file.

***access privileges.*** The type of activities that a database user has permission to perform in a specific database, table, or column. Informix maintains its own set of database access privileges that are independent of the operating system access privileges for system files. Access privileges are sometimes referred to as "access permission."

***active set.*** The collection of rows satisfying a query associated with a cursor.

***after-image.*** The values in a row written into an INFORMIX-OnLine logical log after a change is made to that row.

***aggregate functions.*** The functions that return a single value based on the values of columns in one or more rows of a table. Examples are the total number, sum, average, maximum, or minimum of an expression in a query or a report. Aggregate functions are sometimes referred to as "set functions" or "math functions."

***alias.*** A temporary alternative name for a table in a query. Usually used in complex subqueries. In a form-specification file, alias refers to a single-word alternative name used in place of a more complex table name (for example, *owner.table-name*).

***ANSI.*** Acronym for the American National Standards Institute. This group sets standards for the computer industry, including standards for SQL languages.

***application development tool.*** Software such as INFORMIX-SQL, INFORMIX-4GL, and INFORMIX-ESQL that a developer can use to create and maintain a database. The software allows a user to send instructions and data to and receive information from the database engine.



An application development tool is sometimes referred to as the “front end.”

**application program.** A file or logically related set of files that performs one or more database management tasks.

**archiving.** Copying all the data and indexes of a database onto a new medium, usually a tape or a different physical device from the one that stores the database. INFORMIX-OnLine copies all its data at a single archive, which may require copying multiple databases.

**argument.** A value passed to a function or a command.

**array.** A set of items of the same type. Individual members of the array are referred to as *elements* and are usually distinguished by an integer argument that gives the position of the element in the array. Informix arrays can have up to three dimensions.

**ASCII.** Acronym for the American Standards Committee for Information Interchange. Often used to describe an ordered set of printable and non-printable characters used in computers and telecommunication.

**attribute.** The qualifier for the method of displaying or verifying data in fields of a screen form.

**audit trail.** A history of all changes to a table.

**audit trail log.** A file containing a history of all changes to a table. Starting from an archived database, an audit trail log can reconstruct all subsequent changes to the table.

**authorization.** The permission to execute an action such as a system command.

**B+ tree.** A method of organizing an index for efficient retrieval of records. All Informix database engines use this access method.

**backup.** To make a duplicate of a computer file on another device or on tape in order to preserve existing work in case of a computer failure or other mishap.

**before-image.** The image of a row, page, or other item before any changes are made to it.

**BLOBs.** An acronym for Binary Large Objects, data objects that effectively have no maximum size (theoretically as large as  $2^{31}$  bytes). **INFORMIX-OnLine** supports two BLOB data types: TEXT for storing text data and BYTE for storing any type of binary data.

**blobpage.** The unit of disk allocation within a blobspace in **INFORMIX-OnLine**. The Database Administrator determines the size of a blobpage; the size can vary from blobpage to blobpage.

**blobspace.** A logical collection of blobpages used to store TEXT and BYTE data in **INFORMIX-OnLine**.

**Boolean.** A variable or an expression that can take on the logical values TRUE (1), FALSE (0), or UNKNOWN (if NULL values are involved).

**breakpoint.** A named object, specified by a debugger, that the programmer can associate with a statement, program block, variable, or logical condition. When a breakpoint is reached, program execution is suspended, allowing the programmer to examine the current value of program variables or the execution stack and to optionally execute debugger commands at that point. A breakpoint must be enabled to take effect. See *debugger*.

**buffer.** A portion of computer memory where a program temporarily stores data. Data is typically read into or written out from buffers to disk. **INFORMIX-OnLine** buffers are all multiples of the same size, namely, the size of a page.

**buffered log.** Holds transactions in a memory buffer until the buffer is full regardless of when the transaction is committed or rolled back. **INFORMIX-OnLine** provides this option to speed operations by reducing the number of disk writes.

**byte.** A unit of storage, approximately corresponding to one character. A kilobyte is 1024 bytes. A megabyte is  $10^6$  bytes. (When BYTE appears in uppercase, it refers to the Informix data type.)

**capability.** Codes used in **termcap** or **terminfo** files that specify terminal functions, such as clearing the screen or the size of the screen.

**Cartesian product.** The set that results when you pair each and every member of one set with each and every member of another set. A Cartesian product results from a multiple-table query when you do not specify the joining conditions among tables. See *join*.

**checkpoint.** A periodic time when **INFORMIX-OnLine** writes all buffers in memory to disk so that data on disk matches what is in memory. Checkpoints are marked by a special record written into the logs.

**chunk.** A large continuous section of disk space for **INFORMIX-OnLine**. A chunk often corresponds to a UNIX disk partition. A specified set of chunks defines a dbspace or a blobspace.

**close a cursor.** To drop the association between the active set of rows resulting from a query and a cursor.

**close a database.** To relinquish the “current” status of a database. Only one current database can exist at a time.

**close a file.** To drop the association between a file and a program.

**column.** A data element containing a particular type of information that occurs in every row of the table. Column also can refer to a position on the screen or in a window.

**comment.** Information in a program file, report specification, or screen form that is not processed by the computer but that documents the program. You use special characters, such as a pound sign ( # ) or curly braces ( { } ), to identify comments.

**command file.** A system file containing a sequence of statements or commands.

**COMMIT WORK.** To terminate a transaction by accepting all changes to the database since the transaction began.



**COMMITTED READ.** A level of process isolation available through INFORMIX-OnLine in which a user can view only rows that are currently committed at the moment of the query request; that is, a user cannot view rows that were changed as a part of a currently uncommitted transaction. It is the default level of isolation under INFORMIX-OnLine for non-MODE ANSI databases. See *process isolation*.

**compile.** To translate a file containing instructions (in a higher level language) into a file containing the corresponding machine-level instructions.

**compile-time errors.** Errors that occur at the time the program source code is converted to executable form. See *run-time errors*.

**composite index.** An index constructed on two or more columns of a table. The ordering imposed by the composite index varies least frequently on the first named column and most frequently on the last named column.

**composite join.** A join between two tables based on the relationship among two or more columns in each table.

**concatenate.** To form the character string that results when a second string is appended to the end of a first string.

**concurrency.** The ability of two or more processes to access the same database simultaneously.

**control character.** A character whose occurrence in a particular context initiates, modifies, or stops a control function (an operation to control a device, for example, in moving a cursor or in reading data). In a program, you can define actions that use the CTRL key in conjunction with another key to execute some programming action (for example, entering CTRL-W to obtain on-line help in Informix products). A control character is sometimes referred to as a "control key." See *printable character*.

**constant.** A non-varying data element or value.

**constraint.** See *UNIQUE CONSTRAINT*.



**corrupted database.** A database whose tables or indexes contain incomplete or invalid data following an aborted INSERT, UPDATE, or DELETE.

**corrupted index.** An index that does not correspond exactly to its table.

**cursor.** An identifier associated with a group of rows. Conceptually, the pointer to the current row. You can use cursors for SELECT statements (associating the cursor with the rows returned by a query) or INSERT statements (associating the cursor with a buffer to insert multiple rows as a group). A SELECT cursor is DECLARED for sequential only (regular cursor) or non-sequential (SCROLL cursor) retrieval of row information. In addition, you can DECLARE a SELECT cursor FOR UPDATE (initiating locking control for UPDATED and DELETED rows) or WITH HOLD (completing a transaction will not close the cursor). The term *cursor* also refers to the position indicator on a video terminal.

**CURSOR STABILITY.** A level of process isolation available through INFORMIX-OnLine in which the database engine must secure a shared lock on a FETCHed row before the row can be viewed. The engine retains the lock until it receives a request to FETCH a new row. See *process isolation*.

**current row.** The most recently retrieved row of the active set of a query.

**data integrity.** The process of ensuring that data corruption does not occur when multiple users simultaneously try to alter the same data. Locking and transaction processing are used to control data integrity.

**data type.** A descriptor assigned to each column in a table or program variable indicating the type of data the column or program variable is intended to hold. Informix data types include SMALLINT, INTEGER, SERIAL, SMALLFLOAT, FLOAT, DECIMAL, MONEY, DATE, DATETIME, INTERVAL, CHAR, VARCHAR, TEXT, and BYTE.

**database.** A collection of information (contained in tables) that is useful to a particular organization or used for a specific purpose.

**Database Administrator (DBA).** The individual(s) responsible for the contents and use of a database. The Database Administrator may also be responsible for the particular contents and use of an INFORMIX-OnLine system.

**database application.** A program that applies database management techniques to implement specific data manipulation and reporting tasks.

**database dictionary.** The collection of tables used by the database management programs to keep track of the structure of the database. Information about the database is maintained in the database dictionary, which is sometimes referred to as the "data dictionary" or "system catalogs."

**database engine.** The portion of the database management system that actually manipulates the database files. This is the process that receives SQL statements from the database application, parses them, optimizes the approach to the data, retrieves the data from the database, and returns the data to the application. The database engine is sometimes referred to as the "back end," "server," or "database agent."

**database management system (DBMS).** All the components necessary to create and maintain a database, including the application development tools and the database engine.

**DB-Monitor.** An interface that presents a series of screens through which a Database Administrator can monitor and modify an INFORMIX-OnLine system. Through the DB-Monitor, an administrator can initialize a system, tune shared-memory parameters, archive and restore dbspaces and logs, add new chunks and dbspaces, and observe the current state of the system.

**DBA.** Acronym for Database Administrator.

**dbspace.** A logical collection of chunks that represents a unified region of disk space in INFORMIX-OnLine. In some ways, the dbspace corresponds to a directory in the UNIX file system. For example, you can create a table in a particular dbspace.

**deadlock.** A situation where two or more processes cannot proceed because each is waiting for a lock held by the other (or another) process. INFORMIX-OnLine monitors potential deadlock situations and

prevents them by sending an error message to the process whose request for a lock would result in a deadlock. In distributed queries across multiple systems, INFORMIX-STAR controls deadlocks by having the administrator set a maximum amount of processing time. Distributed queries are aborted at the end of this time. See *multisite deadlock*.

**debugger.** A software product to analyze programs and to detect and locate errors in program logic. The INFORMIX-4GL Interactive Debugger is a 4GL source language debugger that supports a wide variety of programming tools, such as tracing program logic and stopping execution at preset points. See *breakpoint* and *tracepoint*.

**declarative statement.** The programming language statements that describe or define objects, for example, defining a program variable. See *procedural statement*.

**default.** How a program acts if the user does not explicitly specify an action.

**delimiter.** A boundary on an input field or the terminator for a column or row. In a form specification, the field delimiters are square brackets ( [ ] ) and determine the size of the field. Some files and PREPARED objects require semicolon ( ; ) delimiters between statements.

**DIRTY READ.** A level of process isolation that does not account for locks and does allow viewing of any existing rows, even rows that may be currently altered from inside an uncommitted transaction. DIRTY READ is the lowest level of isolation (no isolation at all). It is the level at which INFORMIX-SE operates, and it is an optional level under INFORMIX-OnLine. See *process isolation*.

**disk configuration.** The organization of a disk into several separate regions called partitions.

**disk I/O.** The process of transferring data between memory and disk.

**disk mirroring.** Storing the same data on two disks simultaneously. If one disk crashes, the data is still usable on the remaining disk. This option is available with INFORMIX-OnLine.



***display field.*** Used in a screen form to indicate where data is to be displayed on the screen. A display field is usually associated with a column in a table.

***display label.*** A temporary name for a column or expression in a query.

***distributed option.*** The ability to access data in multiple databases across multiple INFORMIX-STAR systems. The systems may be on the same hardware or on a computer network. INFORMIX-OnLine can perform multiple-database queries within a single system. INFORMIX-STAR can span multiple systems.)

***dominant table.*** See *outer join*.

***dynamic statements.*** SQL statements that are created at the time the program is executed rather than when the program is written. The PREPARE statement in INFORMIX-4GL and INFORMIX-ESQL is used to create dynamic statements.

***embedded SQL.*** SQL statements placed within a host language. Informix supports embedded SQL in C, COBOL, FORTRAN, and Ada.

***environment variable.*** A variable with an assigned value that is maintained by the operating system and made available to all programs.

***error message.*** A message displayed on the screen or written to a file to describe either the failure of an action or an illegal specification. Each error is identified by a (usually negative) designated number.

***error log.*** A file that receives error information whenever a program runs.

***error trapping.*** Code within a program that anticipates and reacts to run-time errors.

***escape key.*** The key (usually marked ESC or ESCAPE) used to terminate one mode and start another in most UNIX and DOS systems. On many terminals the ESCAPE key is the default Accept key (used to indicate when you are finished entering text in a query, add, update, or delete action) for PERFORM and INFORMIX-4GL screen forms.



***exclusive access.*** The user has sole access to the information. Other users are prevented from using the database or table.

***exclusive lock.*** A lock on an object (row, page, table, or database) held by a single process that prevents other processes from acquiring a lock of any kind on the same object.

***executable file.*** A binary file containing compiled code that can be run as a program. May also refer to a UNIX shell script, a DOS batch file, or a VMS command file.

***execute.*** To carry out a program or a set of instructions.

***expansion page.*** An additional page filled with data from a single row. INFORMIX-OnLine uses expansion pages when the data for a row cannot fit in the initial page. Expansion pages are added and filled as needed. The original page entry contains pointers to the expansion page(s). See *home page*.

***expression.*** Anything from a simple numeric or alphabetic constant to a more complex series of column values, functions, quoted strings, operators, and keywords. A Boolean expression contains a logical operator (>, <, =, !=, IS NULL, and so on) and evaluates as TRUE, FALSE, or UNKNOWN. An arithmetic expression contains the operators (+, -, ×, /, and so on) and evaluates as a number.

***extent.*** A continuous segment of disk space allocated to a tblspace in INFORMIX-OnLine. The programmer can specify both the initial extent size for a table and the size of all subsequent extents assigned to the table.

***external, distributed table.*** A database table that is not in the current database and is on a different INFORMIX-STAR system.

***external table.*** A database table that is not in the current database but is on the same INFORMIX-OnLine system.

***fast recovery.*** The feature of INFORMIX-OnLine that automatically restores a database to its state at the end of the last completed transaction after a non-destructive failure of the computer system (for example, a power failure but not a disk failure). Fast recovery uses the physical log to roll back to the last checkpoint, and the logical logs to roll forward from there on.

***fault tolerance.*** See *high availability*.

***file.*** A collection of related information stored together on the system, such as the words in a letter or report, a computer program, or a listing of data.

***filename extension.*** The part of a filename following the period. For example, ACE appends the extension *.arc* to the filename of a compiled report specification.

***flag.*** A command line option, usually indicated by a minus (-) sign in UNIX systems. For example, ACE accepts the *-s* (silent) flag on the command line.

***floating point number.*** A number with fixed precision (total number of digits) and undefined scale (number of digits to the right of the decimal point). The decimal point “floats” as appropriate to represent an assigned value.

***footer.*** See *page trailer*.

***forced residency.*** An option that forces UNIX to keep INFORMIX-OnLine shared memory segments always resident in memory, preventing UNIX from swapping out these segments to disk on a busy system. (This option is not available in all UNIX systems.)

***form specification.*** A system file containing instructions describing how a form looks and performs. You must compile a form specification before you can use it.

***function.*** See *program block*.

***global variable.*** A variable whose value you can access from any module or function in a program. See *variable* and *scope of reference*.

***header.*** See *page header*.

***help message.*** On-line text displayed automatically or at the request of the user to assist the user in interactive programs. Such messages are stored in help files.

**hierarchy.** A tree-like data structure where some groups of data are subordinate to others such that (1) only one group (called *root*) exists at the highest level and (2) each group except *root* is related to only one group on a higher level.

**high availability.** The ability of a system to resist failure and loss of data. High availability includes features such as fast recovery and disk mirroring. It is sometimes referred to as “fault tolerance.”

**highlight.** An inverse-video rectangular area that marks your place on the screen. A highlight often indicates the current option on a menu or the current character in an editing session. If a terminal cannot display highlighting, the current option often appears in angle brackets while the current character is underlined.

**home engine.** The first database engine that the current database accesses in a distributed query across a network of INFORMIX-STAR systems.

**home page.** The original memory address of a row in INFORMIX-OnLine. See *expansion page*.

**host variable.** A C, COBOL, FORTRAN, or Ada program variable that is referenced in a statement. A host variable is identified by the dollar sign ( \$ ) or colon ( : ) that precedes it.

**identifier.** A sequence of letters, digits, and underscores ( \_ ) that represents the name of a database, table, column, screen form, program variable, cursor, function, index, window, menu, synonym, alias, view, PREPARED object, constraint, or report.

**incremental archiving.** A three-level system of archiving in INFORMIX-OnLine that allows you the option to archive only those parts of the data that have changed since the last archive.

**index.** A file containing pointers to rows of data. Indexes can speed ordering of rows and optimize the performance of database queries.

**infocmp.** A UNIX program that you can use to view or decompile **terminfo** files, or that you can use to compare compiled **terminfo** entries with entries in a **termcap** file.



***initialize.*** Assigning a starting value.

***input.*** Information received from an external source (for example, from the keyboard, a file, or another program) and processed by a program.

***installation.*** Loading software from some magnetic medium (tape, cartridge, floppy disk) onto the computer and preparing it for use.

***interactive.*** Programs that accept input from the user, process the input, and then produce output on the screen, in a file, or on a printer.

***interpreter.*** A program that reads, decodes, and executes statements one at a time.

***interrupt.*** A signal from a user or another process that can stop the current process temporarily or permanently. See *signal*.

***ISAM.*** Acronym for Indexed Sequential Access Method. An access method is a way of retrieving pieces of information (rows) from a larger set of information (table). An indexed sequential access method allows you to find information in a specific order or to find specific pieces of information quickly through an index.

***join.*** The process of combining information from two or more tables based on some common domain of information. Rows from one table are paired with rows from another table when information in the corresponding rows match on the joining criterion. For example, if a **customer\_number** column exists in both **customer** and **orders** tables, you can construct a query that pairs each **customer** row with all the associated **orders** rows based on the common **customer\_number**. See *Cartesian product* and *outer join*.

***kernel.*** The part of the UNIX operating system that controls processes and the allocation of resources.

***key.*** The piece(s) of information used to locate a row of data. A key defines the piece(s) of information you want to search for as well as the order in which you want to process information in a table. For example, you can index the **last\_name** column in a **customer** table to find specific customers or to process the customers in alphabetical or reverse alphabetical order according to last name.



**keyword.** A word that has meaning to a program. For example, the word SELECT is a keyword in SQL relating to database queries.

**latch.** A communication device that controls the use of resources. Informix database engines use latches to control access to the lock and other management tables. Informix *latches* are similar in function to UNIX *semaphores*, though the two are separate control devices.

**level of isolation.** See *process isolation*.

**library.** A collection of pre-compiled functions designed to perform tasks common to a particular kind of application. Your product may include library functions that you can call from your programs.

**link.** The process of combining separately compiled program modules into an executable program.

**literal.** A character constant. In the format string of a PICTURE attribute, for example, any characters except A, #, and X are literals because they are displayed exactly as they appear in the format string.

**local variable.** A variable that has meaning only in the routine in which it is defined. See *variable* and *scope of reference*.

**lock mode.** An option that sets whether a user who requests a lock on an already locked object will (1) not wait for the lock and instead get an error, (2) wait until the object is released to get the lock, or (3) wait a certain amount of time before getting an error (an option available only with INFORMIX-OnLine). In INFORMIX-OnLine, the lock mode can also refer to the standard unit of locking (either page or row) chosen by the programmer.

**locking.** The process of temporarily limiting access to an object (database, table, page, or row) to prevent conflicting interactions among concurrent processes. Locks can be in either EXCLUSIVE MODE, which restricts both read and write access to only one user, or SHARE MODE, which allows read-only access to other users. In addition, there are UPDATE locks that begin in SHARE MODE but are upgraded to EXCLUSIVE MODE when a row is actually changed.

**locking granularity.** The size of an object that is locked. The size may be a database, table, page, or row.

**logical log.** A file containing a list of all changes that were performed on a database during the period the log was active. The logical log is used to roll back transactions, recover from system failures, and restore databases from archives. In **INFORMIX-OnLine**, the logical log is also used to roll dbspaces forward from some checkpoint. This log is also referred to as a "transaction log."

**login.** The process of identifying oneself to a computer.

**macro.** A name given to a set of instructions that the computer executes whenever the name is referenced.

**mantissa.** The significant digits in a floating point number, usually expressed as a number between zero and one.

**menu.** A screen display that allows you to choose the commands that you want the computer to perform.

**mirroring.** See *disk mirroring*.

**MODE ANSI.** Refers to a database that conforms to certain ANSI performance standards. Informix databases can be created as either MODE ANSI or non-MODE ANSI. A MODE ANSI database enforces certain ANSI requirements that are not enforced in non-MODE ANSI databases, such as implicit transactions, required owner-naming, and no buffered logging (when using **INFORMIX-OnLine**).

**module.** The part of a program that resides in a single system file. Each file represents one module. Entire programs may reside in a single module, or they may be contained in several modules, each performing a specific function or purpose.

**module variable.** A variable whose value you can reference from any program block within the same module but not from other modules. See *variable* and *scope of reference*.

**monochrome.** Describes a monitor that can display only one color.

**multisite deadlock.** A deadlock that occurs between processes due to a distributed query across multiple **INFORMIX-STAR** systems. **INFORMIX-STAR** controls multisite deadlocks by having the Database Administrator set a maximum processing time, at which point the query is aborted.

**NULL value.** A value representing “unknown” or “not applicable.” (A NULL is not the same as a value of zero or blank.)

**offset.** Used in **INFORMIX-OnLine** to specify the physical position of a defined chunk on a disk. The offset is literally the number of kilobytes indented into the named device (which is the specified disk partition) before starting the chunk.

**open.** The process of making a resource available, such as preparing a file for access, activating a cursor, or initiating a window.

**operating mode.** **INFORMIX-OnLine** states of operation that include the following:

graceful-shutdown	Terminates all user processes upon completion of current activities and goes into quiescent mode.
immediate-shutdown	Terminates all user processes immediately and goes into quiescent mode.
off-line	The beginning mode of an <b>INFORMIX-OnLine</b> system until the administrator initializes the system.
on-line	Normal user access.
quiescent	No user access allowed; only administrative functions can be performed.
recovery	Occurs when the system goes from off-line to quiescent; fast recovery automatically occurs if necessary.
take-offline	Terminates all user processes and immediately goes into off-line mode.

**outer join.** An asymmetric joining of a dominant and a subservient table in a query whereby joining restrictions apply only to the subservient or “outer” table. Rows in the dominant table are retrieved without considering the join, but rows from the outer table are included only if they also satisfy the join conditions. Any dominant-table rows that do not have a matching row from the outer table receive a row of NULLs in place of an outer-table row.



**output.** The result that the computer produces in response to such things as a query or a request for a report.

**pad.** To fill empty places at the beginning or end of a line, string, or field, usually with a space or a blank.

**page.** The basic unit of disk and memory storage used by INFORMIX-OnLine. The page size is two kilobytes for most ports, but it can be one, two, four, or eight kilobytes. The size is fixed for a port, and the customer cannot tune it. Depending on row size, a page may contain a portion of a row, one row, or multiple rows.

**page header.** The items that are printed at the top of each page of a report (for example, the title and date). A “running header” appears at the top of each page of a report.

**page trailer.** The items that are printed at the bottom of each page of a report (for example, the page number). A page trailer is also referred to as a “footer.” A “running footer” appears at the bottom of each page of a report.

**parameter.** A variable that is given a constant value for a specified application. In a subroutine, a parameter commonly uses an argument value passed to that routine.

**pattern.** An identifiable or repeatable series of characters or symbols.

**permission.** The right to use or change the contents of a database. On some operating systems, the right to have access to files and directories.

**phantom row.** A row of a table that is initially modified or inserted during a transaction but subsequently rolled back. Another process may see a phantom row if the isolation level is DIRTY READ. No other isolation level allows a changed but uncommitted row to be seen.

**physical log.** A file in INFORMIX-OnLine that records the before-images of all pages changed since the last checkpoint. The physical log is used only for fast recovery and is emptied at each checkpoint.



***pointer.*** A number that specifies the address in memory of the data or variable of interest.

***pop.*** Removes a value from a stack in memory. See *stack* and *push*.

***precision.*** The total number of significant digits in a real number, both to the right and left of the decimal point. For example, the number 1437.2305 has a precision of 8. See *scale*.

***PREPARED statement.*** An SQL statement that translates a character string created at run time into a request of the database. This feature allows you to form your request while the program is executing without having to modify and recompile the program.

***preprocessor.*** A program that takes high-level programs and produces code that a standard language compiler, such as C, can compile.

***primary key.*** The information from a column or set of columns that uniquely identifies each row in a table. The primary key is sometimes called a "unique key."

***printable character.*** A character that can be displayed on a terminal or printer. Includes A-Z, a-z, 0-9, and punctuation. See *control character*.

***procedural statements.*** The programming language statements that specify actions, for example, looping and branching if a condition is met. See *declarative statement*.

***procedure.*** See *program block*.

***process.*** An individual task performed by the system. Each Informix user typically invokes two processes, one for the application development tool he or she is using and one for the database engine.

***process isolation.*** The level of process independence among multiple users when they attempt to access common data, specifically relating to the locking strategy for read-only SQL requests. The various levels of isolation are distinguished primarily by the length of time that shared locks are (or can be) acquired and held. **INFORMIX-SE** sets a level of no isolation (referred to as a DIRTY READ), and this cannot

be changed. **INFORMIX-OnLine** allows the programmer to choose from four levels of isolation. See *DIRTY READ*, *COMMITTED READ*, *CURSOR STABILITY*, and *REPEATABLE READ*.

**program block.** A named collection of statements that performs a particular task; a unit of program code. In 4GL, it refers to a **MAIN**, **FUNCTION**, **REPORT**, or **GLOBALS** section. A program block is sometimes referred to as a “function,” “procedure,” or “routine” (although these terms in some languages have subtle but distinct differences in meaning).

**program control.** Actions that the computer takes, as opposed to actions that the user takes.

**projection.** A specific set of columns from a table listed in a database query.

**promotable lock.** A lock that can be changed from a shared lock to an exclusive lock. See *update lock*.

**push.** Places a value onto a stack in memory. See *stack* and *pop*.

**query.** A request to the database to retrieve data that meet certain criteria.

**raw device.** A UNIX disk partition that is defined as a character device and that is not mounted. The UNIX file system is unaware of a raw device.

**raw I/O.** The process of transferring data between memory and a raw device. **INFORMIX-OnLine** can bypass the UNIX file system and address raw devices directly. The advantage of raw I/O is that data on a disk can be organized for more efficient access. Raw I/O is sometimes referred to as “direct I/O.”

**record.** See *row*.

**recover a database.** To restore a database to a former condition after a system failure or other destructive event. The recovery restores the database as it existed immediately before the crash. This is sometimes referred to as “restore a database.”

**relation.** See *table*.

**relational database.** A database that uses a table structure to store data. Relationships among tables are logically specified at the time of user access into the database; they are not built into the data structures themselves (unlike some other database systems).

**REPEATABLE READ.** A level of process isolation available through INFORMIX-OnLine that ensures all data read during a transaction is not modified by another process. Transactions under REPEATABLE READ are also known as serializable transactions. It is the default level of isolation under INFORMIX-OnLine for MODE ANSI databases. See *process isolation*.

**report specification.** A file or program segment that contains the description of a report. The report is described in a report-writing language.

**report writer.** A program, such as ACE, that allows the user to describe the appearance of a report, using a report-writing language. The report writer can then compile this report specification into an executable report.

**reserved word.** A word in a statement or command that you cannot use in any other context of the language or program.

**restore a database.** See *recover a database*.

**roll back.** The process that reverses an action or series of actions upon a database. The database is returned to the condition that existed before the statements were executed. See *transaction*.

**roll forward.** The process that brings an archive copy of a database up to date. This process usually takes place when a database is *recovered* after a system crash or other failure. An archive copy of the database is restored to the disk, and the database is rolled forward to a point just before the failure.

**root dbspace.** The initial dbspace for an INFORMIX-OnLine system. In addition to any data, the root dbspace contains all system management tables, the physical log, and at least the initial logical log. A Database Administrator has the option to add other dbspaces and logical logs.



**routine.** See *program block*.

**row.** A group of related items of information about a single entity in a database table. In a table of customer information, for example, a row contains information about a single customer. A row is sometimes referred to as a “record” or “tuple.” (In a screen form, *row* may refer to a line of the screen.)

**run-time errors.** Errors that occur during program execution. See *compile-time errors*.

**scale.** The number of digits to the right of the decimal place in DECIMAL notation. The number 1437.2350 has a scale of 4 (four digits to the right of the decimal point). See *precision*.

**schema.** A listing of the structure of a database or a table. The schema for a table lists the names of the columns, their data types and (where applicable) lengths, indexing, and other information about the structure of the table.

**scope of reference.** The portion of a program in which an identifier applies and can be accessed. There are three possible scope sizes: local (an identifier applies only within a single program block), modular (the identifier applies throughout a single module), and global (an identifier applies throughout the entire program). Identifiers must be declared before you can reference them. For example, a module identifier cannot be referenced prior to the statement within the module that declares it.

**screen form.** A data-entry form that is displayed on the screen of a terminal. The user enters data into the blanks on the form.

**self-join.** A join between a table and itself. A self-join occurs when a table is used two or more times in a SELECT statement (under different aliases) and joined to itself.

**semaphore.** A UNIX communication device that controls the use of system resources. See *latch*.

**serializable transactions.** See *REPEATABLE READ*.



**servername.** The unique name that the Database Administrator assigns to an INFORMIX-OnLine system at the time of initialization. The server name is used to access external tables and databases.

**servernum.** The unique number between 0 and 99 that the Database Administrator assigns to an INFORMIX-OnLine system at the time of initialization. If more than one INFORMIX-OnLine system is installed on the same machine, each system must have a unique number.

**shared lock.** A lock that more than one process can acquire on the same object. Shared locks allow for greater data integrity with multiple users; if two users have locks on a row, a third user cannot change the contents of that row until both users (not just the first) release the lock. Shared-locking strategies are used in all levels of process isolation except DIRTY READ.

**shared memory.** Memory accessible to multiple processes. In non-shared memory systems, different processes cannot access information in each other's data spaces. Shared memory allows multiple processes to access a common data space in memory. Common data does not have to be reread from disk for each process, reducing disk I/O and improving performance.

**signal.** A special character or set of characters used as a means of communication between two processes. For example, signals are sent when a user or a program wishes to interrupt or suspend the execution of a process. See *interrupt*.

**singleton select.** A SELECT statement that returns a single row.

**source file.** A file containing instructions (in ASCII text) that is used as the source for generating compiled programs.

**SQL.** Acronym for Structured Query Language. A database query language developed by IBM and standardized by ANSI. Informix database management products are based on an extended implementation of ANSI-standard SQL.

**stack.** An area of memory reserved for the temporary storage of data elements used by a program. (There can be more than one stack.) The stack usually holds data that is of immediate use to the program, such as the arguments passed to a function. Items are removed from

the stack in the reverse order from which they were inserted. See *push* and *pop*.

***stack operator.*** Operators that allow programs to manipulate values that are on the stack.

***statement.*** A line, or set of lines, of program code that describes a single action (for example, a SELECT statement or an UPDATE statement).

***status variable.*** A program variable that indicates the status of some aspect of program execution. Status variables often store error numbers or act as flags to indicate that an error has occurred.

***string.*** A set of characters (generally alphanumeric) that is manipulated as a single unit. A string might consist of a word (such as "Smith"), a set of digits representing a number (such as "19543"), or any other collection of characters.

***subquery.*** A query that is embedded as part of another SQL statement. For example, an INSERT statement may contain a subquery in which a SELECT statement supplies the inserted values in place of a VALUES clause; an UPDATE statement may contain a subquery in which a SELECT statement supplies the updating values; or a SELECT statement may contain a subquery in which a second SELECT statement supplies the qualifying conditions of a WHERE clause for the first SELECT statement. (Parentheses usually, but not always, delimit a subquery.)

***subservient table.*** See *outer join*.

***synonym.*** A name assigned to a table and used in place of the original name for that table. A synonym does not replace the original table name; instead, it acts as an alias for the table.

***system catalog.*** A database table that contains information about the database itself, such as the names of tables or columns in the database, the number of rows in a table, information about indexes and database privileges, and so forth.

***system log.*** The UNIX file that INFORMIX-OnLine keeps to record significant events like checkpoints, filling of log files, recovery data, and errors.



**table.** A rectangular array of data in which each row describes a single entity and each column contains the values for each category of description. For example, a table might contain the names and addresses of customers. Each row corresponds to a different customer, and the columns correspond to the name and address items. A table is sometimes referred to as a "relation."

**tbconfig file.** A file that INFORMIX-OnLine uses on initialization which contains configuration data. Administrators who use the DB-Monitor do not need to work with **tbconfig** directly, except to keep track of the maximum values of the initialization parameters.

**tblspace.** The logical collection of extents assigned to a table in INFORMIX-OnLine.

**temporary.** An attribute of any file, index, or table that is deleted when program execution terminates.

**termcap.** An ASCII file that contains the names and capabilities of common terminals.

**terminfo.** A hierarchical directory structure that contains compiled files of terminal capabilities.

**tic.** A UNIX program that compiles **terminfo** source files or **terminfo** files that have been decompiled using **infocmp**. See *infocmp*.

**timeout.** The point at which a lock request is aborted because the requesting process has waited longer for the lock than the specified maximum time limit. A program developer can set a time limit in INFORMIX-OnLine through the SET LOCK MODE statement, and the Database Administrator sets a time limit for operations across multiple INFORMIX-STAR systems.

**tracepoint.** A named object, specified by a debugger, that the programmer can associate with a statement, program block, or variable. When the tracepoint is reached, the debugger displays information about the associated statement, program block, or variable and executes any optional commands that are specified by the programmer. A tracepoint must be enabled to take effect. See *debugger*.

**transaction.** A collection of one or more SQL statements that is treated as a single unit of work. If one of the statements in a transaction fails, the entire transaction can be *rolled back* (canceled). If the transaction is successful, the work is *committed* and all changes to the database from the transaction are accepted.

**transaction log.** See *logical log*.

**tuple.** See *row*.

**UNIQUE CONSTRAINT.** A specification that a database column (or composite list of columns) cannot contain two rows with identical values. You can assign a name to a constraint.

**unique key.** See *primary key*.

**unlock.** To free an object (database, table, page, or row) that has been locked. For example, a locked table prevents others from adding, removing, updating, or (in the case of an exclusive lock) viewing rows in that table as long as it is locked. When the user or program unlocks the table, others are again permitted access.

**update lock.** A promotable lock acquired during a SELECT... FOR UPDATE. An update lock behaves like a shared lock until the update actually occurs, then it becomes an exclusive lock. It differs from a shared lock in that only one update lock can be acquired on an object at a time.

**user interface.** The part of the program that communicates with the user by prompting for input and displaying output based on information the user enters. Typical user interfaces include menus, prompts, screen forms, and on-line help messages.

**variable.** The identifier for a location in memory that stores the value of a program object whose value can change during the execution of the program.

**view.** A dynamically controlled picture of the contents in a database that allows you to determine what information the user sees and manipulates. A view represents a virtual table based on a specified SELECT statement.



***virtual column.*** A derived column of information that is not stored in the database. For example, you can create virtual columns in a SELECT statement by arithmetically manipulating a single column, such as multiplying existing values by a constant, or by combining multiple columns, such as adding the values from two columns.

***wildcard.*** A special symbol that represents either any sequence of zero or more characters or any single character. In SQL, for example, the asterisk ( \* ), question mark ( ? ), brackets ( [ ] ), percent sign ( % ), and underscore ( \_ ) can be used as wildcard characters. (The asterisk, question mark, and brackets are also wildcards in UNIX.)

***window.*** A rectangular area on the screen in which you can take actions without leaving the context of the background program.

# Error Messages



This section contains the text of error messages that may appear when you work with INFORMIX-ESQL/C and suggests corrective actions. Unless otherwise specified, the statement that generated the error was not processed.

- All INFORMIX-ESQL/C errors are returned by number in `sqlca.sqlcode`. These numbers range from -200 to -1300.
  - Error messages used by products running on the INFORMIX-OnLine database engine range from -528 through -803.
  - The error message numbers for INFORMIX-NET products range from -905 through -956.
- The ISAM errors are returned in `sqlca.sqlerrd[1]`. These error message numbers range from -100 through -149.

Use the numbers in the following list to locate the error description and a possible recovery response:

- 100    **Description of Error:** ISAM error: there is already a record with the same value in a unique key.

**Corrective Action:** Check that you did not attempt to add a duplicate value to a column with a unique index by way of *iswrite*, *isrewrite*, *isrewcurr*, or *isaddindex*.

- 101    **Description of Error:** ISAM error: file is not open.

**Corrective Action:** Check that the ISAM file has been opened using the *isopen* call, or that you have not tried to write to a ISAM file opened for read only.

- 102    **Description of Error:** ISAM error: illegal argument to ISAM function.

**Corrective Action:** Check that one of the arguments to the ISAM call is not outside the range of acceptable values for that argument.



- 103 **Description of Error:** ISAM error: illegal key descriptor (too many parts or too long).

**Corrective Action:** Check that one or more of the elements that make up the key description is not outside the range of acceptable values for that element. (There is a maximum of 8 parts and 120 characters to each key descriptor.)

- 104 **Description of Error:** ISAM error: too many files open.

**Corrective Action:** The maximum number of files that can be open at one time would be exceeded if this request were processed. Reduce the number of open files by breaking your program up into two or more parts. The maximum number of open files per process is 20.

- 105 **Description of Error:** ISAM error: bad ISAM file format.

**Corrective Action:** The format of the ISAM file has been corrupted. Run the **bcheck** utility on the file, and it will try to repair damaged indexes. If **bcheck** cannot repair the file, you must reload your data from a backup medium.

- 106 **Description of Error:** ISAM error: non-exclusive access.

**Corrective Action:** You must first open the file with exclusive access when you want to add or delete an index.

- 107 **Description of Error:** ISAM error: record is locked.

**Corrective Action:** The record or file requested by this call cannot be accessed because it has been locked by another user. Wait a moment and re-enter your request.

If you are certain that the table is not in use, you may need to empty the contents of the *tablename.lok* file. (This file contains information about which rows of the table are being used at any one time. It is normally emptied when a user finishes accessing the table. On occasion, the file is not emptied and, as a result, no one is able to use the table.) You can copy the "file" `/dev/null` into this file to remove all locks on rows in the table. Contact your System Administrator about this action.

- 108 **Description of Error:** ISAM error: key already exists.

**Corrective Action:** You have attempted to add an index that previously has been defined. You must delete the existing index before defining another.

- 109 **Description of Error:** ISAM error: the key is the primary key of the file.

**Corrective Action:** An attempt was made to delete the primary key column. The primary key cannot be deleted by the *isdelindex* call.

- 110 **Description of Error:** ISAM error: end or beginning of the file.

**Corrective Action:** You have reached the beginning or end of the file.

- 111 **Description of Error:** ISAM error: no record found.

**Corrective Action:** No record could be found that contained the requested value in the specified position. Edit your request and re-enter.

- 112 **Description of Error:** ISAM error: there is no current record.

**Corrective Action:** An attempt to access a record in the current list has been made, but there is no current list. You must first perform a query and obtain a current list.

- 113 **Description of Error:** ISAM error: the file is locked.

**Corrective Action:** The table you want to alter is currently in use by another user in EXCLUSIVE MODE. Wait until the table is no longer being used before entering your request.

If you are certain that the table is not in use, run the UNLOCK TABLE command to unlock the table. Also, if your system contains *tablename.lok* files, you may need to empty the contents of the *tablename.lok* file. (This file contains information about which rows of the table are being used at any one time. It is

normally emptied when a user finishes accessing the table. On occasion, the file is not emptied and, as a result, no one is able to use the table.) You can copy the file `/dev/null` into this file to remove all locks on rows in the table. Be certain no processes are accessing the locked table before emptying the *tablename.lock* file. Contact your System Administrator about this action.

- 114    **Description of Error:** ISAM error: the filename is too long.

**Corrective Action:** Reduce the filename length to 10 or fewer characters.

- 116    **Description of Error:** ISAM error: cannot allocate memory.

**Corrective Action:** Insufficient memory is available to run your request. (INFORMIX-ESQL/C has run out of addressable data space.) Reduce the complexity of your statement or form.

- 118    **Description of Error:** ISAM error: cannot read transaction log record.

**Corrective Action:** Run the **dblog** utility to determine which record contains the problem.

- 119    **Description of Error:** ISAM error: bad log record.

**Corrective Action:** Run the **dblog** utility to determine which record contains the problem.

- 120    **Description of Error:** ISAM error: cannot open log file.

**Corrective Action:** Make sure that the file exists, that you are using the correct pathname, and that you have the appropriate permissions to use the file.

- 121    **Description of Error:** ISAM error: cannot write log file record.

**Corrective Action:** Check that you have the appropriate file permissions.



- 122    **Description of Error:** ISAM error: BEGIN WORK encountered in a database without transactions.
- Corrective Action:** Make sure that you have CREATED or STARTed your database with transactions.
- 123    **Description of Error:** ISAM error: no shared memory.
- Corrective Action:** Check that the Database Administrator has set up the shared memory partition.
- 124    **Description of Error:** ISAM error: no BEGIN WORK found.
- Corrective Action:** You must execute BEGIN WORK before COMMIT WORK or ROLLBACK WORK.
- 125    **Description of Error:** ISAM error: cannot use NFS.
- Corrective Action:** Do not attempt to access remote files on the network using NFS (network file server).
- 126    **Description of Error:** ISAM error: bad row id
- Corrective Action:** Run **bcheck** (on INFORMIX-SE) to check and repair index structures. Run **tbcheck** (on INFORMIX-OnLine) to check and repair index and data structures.
- 127    **Description of Error:** ISAM error: no primary key.
- Corrective Action:** Run **bcheck** to determine the source of the problem. Repair your table if necessary.
- 128    **Description of Error:** ISAM error: no logging.
- Corrective Action:** You have attempted to execute a statement that requires logging. Start your database with transactions and reexecute the statement.



- 129    **Description of Error:** ISAM error: too many users.
- Corrective Action:** The shared memory parameter for the maximum number of users was exceeded. Either ask your System Administrator to adjust the parameter, or try the statement again later.
- 130    **Description of Error:** ISAM error: dbspace not found.
- Corrective Action:** Check the name of your dbspace. Run the DB-Monitor to view existing dbspaces.
- 131    **Description of Error:** ISAM error: no free disk space.
- Corrective Action:** There is not enough contiguous disk space to complete the operation. Consult your System Administrator for assistance.
- 132    **Description of Error:** ISAM error: the row size is too big.
- Corrective Action:** Make the fields of the table smaller or create multiple tables with fewer fields.
- 133    **Description of Error:** ISAM error: audit trail exists.
- Corrective Action:** You cannot cluster a file with an audit trail. If you want to cluster the file, you must first drop the audit trail.
- 134    **Description of Error:** ISAM error: no more locks.
- Corrective Action:** The shared memory parameter for the maximum number of locks was exceeded. Either lock the entire table, wait until more locks are available, or ask your System Administrator to adjust the parameter.
- 135    **Description of Error:** ISAM error: tblspace does not exist.
- Corrective Action:** Check whether the tblspace number is correct. (INFORMIX-OnLine)

- 136    **Description of Error:** ISAM error: no more extents.  
**Corrective Action:** Use the DB-Monitor to add more disk space. (INFORMIX-OnLine)
- 137    **Description of Error:** ISAM error: chunk table overflow.  
**Corrective Action:** There are too many chunks in the dbspace. Reinitialize INFORMIX-OnLine to allow for more chunks. (INFORMIX-OnLine)
- 138    **Description of Error:** ISAM error: dbspace table overflow.  
**Corrective Action:** There are too many dbspaces. Reinitialize INFORMIX-OnLine to allow for more dbspaces. (INFORMIX-OnLine)
- 139    **Description of Error:** ISAM error: logfile table overflow.  
**Corrective Action:** There are too many logfiles. Reinitialize INFORMIX-OnLine to allow for more logfiles. (INFORMIX-OnLine)
- 141    **Description of Error:** ISAM error: tblspace table overflow.  
**Corrective Action:** There are too many tblspaces currently in use. Wait and try later or reinitialize INFORMIX-OnLine to allow for more tblspaces. (INFORMIX-OnLine)
- 142    **Description of Error:** ISAM error: overflow of tblspace page.  
**Corrective Action:** The keys for the table are too large for a page. Reduce the size of the indexes. (INFORMIX-OnLine)
- 143    **Description of Error:** ISAM error: deadlock detected.  
**Corrective Action:** A deadlock has occurred. Roll back the current transaction and retry it.

**-144 Description of Error:** ISAM error: key value locked.

**Corrective Action:** Wait and retry action. (INFORMIX-OnLine)

**-145 Description of Error:** ISAM error: system does not have disk mirroring.

**Corrective Action:** If you are trying to mirror a dbspace, first reinitialize INFORMIX-OnLine with a mirror for the root dbspace. Then mirror other dbspaces. (INFORMIX-OnLine)

**-146 Description of Error:** ISAM error: the other copy of this disk is currently disabled.

**Corrective Action:** Bring up the other chunk of the mirror pair before you bring down this chunk. (INFORMIX-OnLine)

**-147 Description of Error:** ISAM error: archive in progress.

**Corrective Action:** Check that the action you are taking is compatible with archiving. You cannot add a log or mirror a dbspace which contains a log if an archive is in progress. (INFORMIX-OnLine)

**-148 Description of Error:** ISAM error: dbspace is not empty.

**Corrective Action:** Drop all tables from a dbspace before trying to remove it. (INFORMIX-OnLine)

**-149 Description of Error:** ISAM error: INFORMIX-OnLine daemon is no longer running.

**Corrective Action:** You may have killed the OnLine daemon by accidentally killing the wrong process. You must reinitialize INFORMIX-OnLine.



**-200 Description of Error:** Identifier is too long.

**Corrective Action:** Identifiers must be no longer than 18 characters. Select a new identifier of the appropriate length.

**-201 Description of Error:** A syntax error has occurred.

**Corrective Action:** Check that you have not misspelled an SQL statement, placed keywords out of sequence, or included an INFORMIX-ESQL/C reserved word in your query.

**-202 Description of Error:** An illegal character has been found in the statement.

**Corrective Action:** Remove the illegal character (often a non-printable control character) and resubmit the statement.

**-203 Description of Error:** An illegal integer has been found in the statement.

**Corrective Action:** Integers must be whole numbers from -2,147,483,647 to 2,147,483,647. Check that you have not included a number with a fractional portion or a number outside of the range of acceptable whole numbers. Check also that you have not inadvertently entered a letter in place of a number (for example, 125O3 instead of 12503).

**-204 Description of Error:** An illegal floating-point number has been found in the statement.

**Corrective Action:** Check that you have not inadvertently entered a letter in place of a number (for example, 125O3 in place of 12503).

**-205 Description of Error:** Cannot use ROWID for views.

**Corrective Action:** Restructure your statement so that the virtual column is not used to define the view.

**-206 Description of Error:** The specified table name is not in the database.



**Corrective Action:** Check the spelling of the table name in your statement. Check the **systables** system catalog for a list of all database tables.

- 207    **Description of Error:** Cannot update cursor declared on more than one table.

**Corrective Action:** Check that you have not attempted to use a **FOR UPDATE** clause with cursors on multiple tables. Restructure your update statement, perhaps using multiple cursors.

- 208    **Description of Error:** Memory allocation failed during query processing.

**Corrective Action:** Reduce the complexity of your query or program.

- 209    **Description of Error:** Incompatible database format.

**Corrective Action:** You are attempting to use **INFORMIX-ESQL/C** with a database built for an earlier version of **INFORMIX-ESQL/C**. Run **dbupdate** on your database. It prepares the database for your current version of **INFORMIX-ESQL/C**.

- 210    **Description of Error:** Pathname too long.

**Corrective Action:** **INFORMIX-ESQL/C** accepts a pathname of up to 70 total characters. Reduce the length of the pathname.

- 211    **Description of Error:** Cannot read system catalog *catalog-name*.

**System Action Taken:** See below for a list of system actions.

**Corrective Action:** Check the **ISAM** error for information about the source of the problem. Depending upon the content of the statement and the system catalog cited in the error message, the following actions have occurred:

For a CREATE TABLE statement: the **systabauth** catalog was not read, the table was created, but no authorizations are granted to PUBLIC.

For a DROP TABLE statement: if the **systables** catalog was not read, then no action was taken; if the **sysviews** catalog was not read, then the table was dropped but any views built on the table may not have been dropped.

For a DROP VIEW statement: the **sysviews** catalog was not read, and no action was taken.

For a DROP INDEX statement: the **sysindexes** or **systables** catalog was not read, and the index was not dropped.

For a DROP SYNONYM statement: the **systables** catalog was not read (for **tabtype** = S) and the synonym was not dropped.

For a DROP DATABASE statement: the **systables** catalog was not read, and the database was not dropped.

For a START DATABASE statement: the **systables** catalog was not read, and the database was not started.

For a DATABASE statement: the **systables** or **sysusers** catalog was not read, and the database was not selected.

**-212 Description of Error:** Cannot add index.

**Corrective Action:** Check the ISAM error for information about the source of the problem.

**-213 Description of Error:** Statement interrupted by user.

**Corrective Action:** INFORMIX-ESQL/C has received an interrupt signal (probably due to the user pressing the DEL key). Resubmit your statement.

**-214 Description of Error:** Cannot remove file for table *table-name*.

**System Action Taken:** If this is a DROP DATABASE statement, then some tables may have been dropped from the database. If this is a DROP TABLE statement, then some system entries for the table may have been dropped from the database.

**Corrective Action:** INFORMIX-ESQL/C cannot remove one or more of the system catalogs that make up a table. Check the ISAM error for information about the source of the problem. Check with the System Administrator about remedial actions.

**-215 Description of Error:** Cannot open file for table *table-name*.

**Corrective Action:** Check the ISAM error for information about the source of the problem.

**-216 Description of Error:** Cannot remove ISAM index on file.

**Corrective Action:** Check the ISAM error for information about the source of the problem.

**-217 Description of Error:** Column *column-name* not found in any table in the query.

**Corrective Action:** Correct the spelling of the column name or check that column name exists in the database table. Check for the presence of required commas and quotes.

**-218 Description of Error:** Synonym *name* not found.

**Corrective Action:** Check the spelling of the synonym. If needed, check the **syssynonyms** system catalog for a list of available synonyms.

**-219 Description of Error:** Wildcard matching used with a non-character data type.

**Corrective Action:** Wildcards ( \*, ? ) and characters enclosed in brackets [ ] can be used only with CHAR data types. Check the data type of the column.



-220 **Description of Error:** There is no FROM clause in the query.

**Corrective Action:** You must include a FROM clause in the query. Check that you do not have an illegal character (\$, #, &, or a CONTROL character) in the line prior to the FROM keyword.

-221 **Description of Error:** Cannot build temporary file for new table *table-name*.

**Corrective Action:** ISAM cannot access the /tmp directory or the disk may be out of space. Check the ISAM error for information about the source of the problem.

-222 **Description of Error:** Cannot write to temporary file for new table *table-name*.

**Corrective Action:** The disk may be out of space. Check the ISAM error for information about the source of the problem.

-223 **Description of Error:** Duplicate table name *table-name* in the FROM clause.

**Corrective Action:** Remove the redundant table name from the statement or use an alias to rename one of the tables.

-224 **Description of Error:** Cannot open log file.

**Corrective Action:** Check the ISAM error for information about the source of the problem.

-225 **Description of Error:** Cannot create file for system catalog *catalog-name*.

**System Action Taken:** The CREATE DATABASE statement was not completed. Some system catalogs may have been created.

**Corrective Action:** Check the ISAM error for information about the source of the problem.



-226 **Description of Error:** Cannot create index for system catalog *catalog-name*.

**System Action Taken:** The CREATE DATABASE statement was not completed. Some system catalogs may have been created.

**Corrective Action:** Check the ISAM error for information about the source of the problem.

-227 **Description of Error:** Cannot use ORDER BY clause when selecting into temporary table.

**Corrective Action:** Remove the ORDER BY clause from your statement. Place an index on the column you want to order by after creating the temporary table.

-228 **Description of Error:** Cannot have negative characters.

**Corrective Action:** Check that you have not included a negative CHAR data type (for example, -a or -p) in your statement.

-229 **Description of Error:** Cannot open or create a temporary file.

**Corrective Action:** Check the ISAM error for information about the source of the problem.

-230 **Description of Error:** Cannot read a temporary file.

**Corrective Action:** Check the ISAM error for information about the source of the problem.

-231 **Description of Error:** Cannot perform an aggregate function with a DISTINCT on the expression.

**Corrective Action:** Select the expression into a temporary table and then perform an aggregate distinct on the temporary table.

- 232    **Description of Error:** Attempt to update SERIAL column *column-name*.
- Corrective Action:** The values appearing in a SERIAL column are provided by INFORMIX-ESQL/C and cannot be updated.
- 233    **Description of Error:** Cannot read record that is locked by another user.
- Corrective Action:** Another user has locked the record. Wait a moment and re-enter your request.
- 234    **Description of Error:** Cannot insert into virtual column *column-name*.
- Corrective Action:** The specified column is derived from an expression or an aggregate function. Redefine the view.
- 235    **Description of Error:** Character column size too big. The maximum size is 32,511.
- Corrective Action:** Redefine the size of the column to a maximum of 32,511 characters.
- 236    **Description of Error:** Number of columns in INSERT does not match number of VALUES.
- Corrective Action:** Check that the number of columns in the table or in the column list matches the number of values in the VALUES clause or the SELECT clause.
- 237    **Description of Error:** Cannot begin work.
- Corrective Action:** Check the ISAM error number for information about the source of the problem. Contact your System Administrator or Database Administrator if you need help interpreting the ISAM error number.

**-238 Description of Error:** Cannot COMMIT WORK.

**Corrective Action:** Your log file may be corrupted. Check the ISAM error number for information about the source of the problem. Contact your System Administrator or Database Administrator if you need help interpreting the ISAM error number.

**-239 Description of Error:** Cannot insert new row—duplicate value in a UNIQUE INDEX column.

**Corrective Action:** The row contains a value that already exists in the column (indexed as unique) of an existing row. Enter a new value for the column or remove the unique index on the column.

**-240 Description of Error:** Cannot delete a row.

**Corrective Action:** Check the ISAM error for information about the source of the problem.

**-241 Description of Error:** Cannot ROLLBACK WORK.

**Corrective Action:** Check the ISAM error number for information about the source of the problem. Contact your System Administrator or Database Administrator if you need help interpreting the ISAM error number.

**-242 Description of Error:** Cannot open database table *table-name*.

**Corrective Action:** Check the ISAM error for information about the source of the problem.

**-243 Description of Error:** Cannot position within a table *table-name*.

**Corrective Action:** Check the ISAM error for information about the source of the problem.



- 244    **Description of Error:** Cannot do a physical-order read to fetch next row.
- Corrective Action:** Check the ISAM error for information about the source of the problem.
- 245    **Description of Error:** Cannot position within a file by way of an index.
- Corrective Action:** Check the ISAM error for information about the source of the problem.
- 246    **Description of Error:** Cannot do an indexed read to get the next row.
- Corrective Action:** Check the ISAM error for information about the source of the problem.
- 247    **Description of Error:** ROLLFORWARD database failed.
- Corrective Action:** Check the ISAM error for information about the source of the problem. Contact your System Administrator or Database Administrator if you need help interpreting the meaning of the ISAM error.
- 248    **Description of Error:** Value of the host variable in the WHERE clause is NULL.
- Corrective Action:** Use IS [NOT] NULL for NULL operation or initialize the host variable.
- 249    **Description of Error:** Virtual column has no explicit name.
- Corrective Action:** When selecting into a temporary table or creating a view, each temporary or view column based on an expression must be given a unique name. Check that distinct names are provided.
- 250    **Description of Error:** Cannot read record from file for update.



**Corrective Action:** The record may be locked by another user. Check the ISAM error for information about the source of the problem. Column number *number* too big.

- 251 **Description of Error:** Order BY column number too big.

**Corrective Action:** The number of the column noted in your ORDER BY or GROUP BY statement exceeds the total number of columns in the SELECT statement.

- 252 **Description of Error:** Cannot get system information for table.

**System Action Taken:** Some statistics may be updated.

**Corrective Action:** Check the ISAM error for information about the source of the problem.

- 253 **Description of Error:** Identifier too long—maximum length is 18.

**Corrective Action:** Check the spelling or length of the table name.

- 254 **Description of Error:** Too many or too few host variables given.

**Corrective Action:** Check that the number of host variables in the fetch is equal to the number used to define the cursor.

- 255 **Description of Error:** Statement not in a transaction.

**Corrective Action:** The statement must be executed within a transaction. Start a transaction, then perform the statement.

- 256 **Description of Error:** Transaction not available.

**Corrective Action:** INFORMIX-ESQL/C cannot perform a transaction operation (BEGIN WORK, ROLLBACK WORK, COMMIT WORK) on the database because a transaction log was never created for the database. Ask your Database Administrator to create a transaction log for the database.

- 257    **Description of Error:** System limit on cursors exceeded; maximum is *num*.
- Corrective Action:** Reduce the number of cursors used in the program.
- 258    **Description of Error:** System error—invalid statement ID received by the *sqlexec* process.
- Corrective Action:** Notify the Technical Support Department.
- 259    **Description of Error:** Cursor not open.
- Corrective Action:** Open a cursor, then attempt the fetch.
- 260    **Description of Error:** Cannot execute a *SELECT* statement that is *PREPARED*—must use cursor.
- Corrective Action:** Use a cursor for the *SELECT* statement.
- 261    **Description of Error:** Cannot create a file for table *table*.
- Corrective Action:** Check the *ISAM* error for information about the source of the problem.
- 262    **Description of Error:** There is no current cursor.
- Corrective Action:** You cannot perform an *UPDATE* or *DELETE* action when no current row exists. Perform a fetch, then attempt this action.
- 263    **Description of Error:** Cannot lock row for *UPDATE*.
- Corrective Action:** Check the *ISAM* error for information about the source of the problem.
- 264    **Description of Error:** Cannot write to a temporary file.
- Corrective Action:** Check the *ISAM* error for information about the source of the problem.

-265    **Description of Error:** Load or insert cursors must be run within a transaction.

**Corrective Action:** You must start a transaction with a BEGIN WORK before using insert cursors.

-266    **Description of Error:** There is no current row for UPDATE/DELETE cursor.

**Corrective Action:** You cannot perform an UPDATE or DELETE action when no current row exists. Perform a fetch, then attempt this action.

-267    **Description of Error:** The cursor has been previously released and is unavailable.

**Corrective Action:** Make sure that the cursor is open before you make reference to it.

-268    **Description of Error:** Cannot use SELECT DISTINCT with UNION ALL.

**Corrective Action:** Rewrite your statement.

-269    **Description of Error:** Cannot add a column *column-name* that does not accept nulls.

**Corrective Action:** Rewrite your statement.

-270    **Description of Error:** Cannot position within a temporary file.

**Corrective Action:** Check the ISAM error for information about the source of the problem.

-271    **Description of Error:** Cannot insert new row into the table.

**Corrective Action:** Check the ISAM error for information about the source of the problem.



**-272 Description of Error:** No SELECT permission.

**Corrective Action:** Request permission to SELECT from the owner of the table.

**-273 Description of Error:** No UPDATE permission.

**Corrective Action:** Request permission to UPDATE from the owner of the table.

**-274 Description of Error:** No DELETE permission.

**Corrective Action:** Request permission to DELETE from the owner of the table.

**-275 Description of Error:** No INSERT permission.

**Corrective Action:** Request permission to INSERT from the owner of the table.

**-276 Description of Error:** Cursor not found.

**Corrective Action:** You have called for a cursor that has not been DECLARED in the current session. (The current session runs from the issuance of a DATABASE statement to a CLOSE DATABASE statement, or the issuance of the next DATABASE statement.) Declare your cursor within the current (database) session.

**-277 Description of Error:** UPDATE table *table-name* is not the same as the cursor table.

**Corrective Action:** Either declare a cursor on the table used in the UPDATE statement, or update the table used in the cursor. Check the spelling of the table name and cursor name.

**-278 Description of Error:** Too many ORDER BY columns.

**Corrective Action:** Reduce the number of columns included in the ORDER BY clause to eight or fewer.



- 279 **Description of Error:** Cannot GRANT or REVOKE database privileges for table or view.

**Corrective Action:** Database privileges (Connect, Resource, and DBA) cannot be granted on individual tables. Consult the section "User Status and Privileges" in Chapter 1 for appropriate SQL table privilege statements. Total size of ORDER BY columns exceeds 120 bytes.

- 280 **Description of Error:** A quoted string exceeds 256 bytes.

**Corrective Action:** Reduce the number of columns included in the ORDER BY clause so that the total number of characters is less than or equal to 120 (perhaps delete a CHAR column of 30 or more characters).

- 281 **Description of Error:** Cannot add index to a temporary table.

**Corrective Action:** Check the ISAM error for information about the source of the problem.

- 282 **Description of Error:** Found a quote for which there is no matching quote.

**Corrective Action:** Check that all quoted strings are properly terminated with a quote.

- 283 **Description of Error:** Found a non-terminated comment ("{" with no matching "}").

**Corrective Action:** Check that all comments are properly enclosed in braces. (Comments cannot be nested.)

- 284 **Description of Error:** A subquery has not returned exactly one row.

**Corrective Action:** Check data for the subquery. Restructure the subquery by adding more components in the WHERE clause so that only one value is returned.

- 285    **Description of Error:** Invalid cursor received by `sqlexec`.
- Corrective Action:** Issue a `PREPARE` statement within the current session, then re-enter your original statement.
- 286    **Description of Error:** Expression cannot include `ANY` or `ALL`.
- Corrective Action:** `ANY` and `ALL` can only be used in association with a subquery.
- 287    **Description of Error:** Cannot add `SERIAL` column *column-name* to the table.
- Corrective Action:** A `SERIAL` column does not accept `NULL` values. Add the column to the table as type `INTEGER`, `UPDATE` it so that there are no `NULLS`, and then `MODIFY` it to type `SERIAL`.
- 288    **Description of Error:** Table *table-name* not locked by current user.
- Corrective Action:** You attempted to unlock a table that you did not lock.
- 289    **Description of Error:** Cannot lock table *table-name* in `SHARE MODE`.
- Corrective Action:** The table is already locked in `EXCLUSIVE MODE`. Wait until the table is unlocked before re-executing your request.
- 290    **Description of Error:** Cursor not declared with `FOR UPDATE` clause.
- Corrective Action:** You have attempted to delete or update `WHERE CURRENT OF` without declaring the cursor for update. `DECLARE` your cursor `FOR UPDATE`.

- 291    **Description of Error:** Table *table-name* is already locked.
- Corrective Action:** You must unlock the table before executing your request.
- 292    **Description of Error:** An implied INSERT column *column-name* does not accept NULLs.
- Corrective Action:** INFORMIX-ESQL/C does not allow you to insert a NULL value in a column that does not allow NULLS. Check to see if a non-NULL column is included in the column list in the INSERT statement.
- 293    **Description of Error:** IS [NOT] NULL predicate used with other than simple columns.
- Corrective Action:** Restructure your query.
- 294    **Description of Error:** The column *column-name* not in the GROUP BY list.
- Corrective Action:** All non-aggregate columns in the SELECT list must be included in the GROUP BY list. Restructure your statement to include all columns that are not aggregate functions.
- 295    **Description of Error:** The GROUP BY column number is too large.
- Corrective Action:** The number of the column noted in your ORDER BY or GROUP BY statement exceeds the total number of columns in the SELECT statement.
- 297    **Description of Error:** The SELECT list contains a subquery.
- Corrective Action:** Remove the subquery from the SELECT list in the statement. COUNT(DISTINCT) *colname*) may be used only with a simple column.



- 298    **Description of Error:** COUNT(DISTINCT ...) may be used only with a simple column.
- Corrective Action:** You cannot include expressions with the COUNT(DISTINCT ...) function. Restructure the query.
- 299    **Description of Error:** Query contains more than one DISTINCT.
- Corrective Action:** Restructure your query to include only one DISTINCT.
- 300    **Description of Error:** There are too many GROUP BY columns.
- Corrective Action:** Reduce the number of columns in the statement to eight or fewer.
- 301    **Description of Error:** The total size of the GROUP BY columns is too big.
- Corrective Action:** The total number of characters in all columns listed in the GROUP BY list exceeds 120 characters. Reduce the column list (perhaps exclude any columns with character data types greater than 30 characters).
- 302    **Description of Error:** No GRANT option or illegal option on multi-table view.
- Corrective Action:** You do not have permission to grant access privileges to the table. Only the table owner or a user with GRANT permission can do this.
- 303    **Description of Error:** Expression mixes columns with aggregates.
- Corrective Action:** Restructure your query so that columns and aggregates are not included in the same expression.



- 304    **Description of Error:** HAVING can only have expressions with aggregates or columns in GROUP BY clauses.
- Corrective Action:** The HAVING clause can only take on aggregates. Restructure your query.
- 305    **Description of Error:** Subscripted column *column-name* is not of type CHAR, VARCHAR, TEXT nor BYTES.
- Corrective Action:** Remove the subscript delimiter from the non-character column in the request.
- 306    **Description of Error:** Subscript out of range.
- Corrective Action:** The range of the subscript delimiter exceeds the range of the column data type. Check the size of the data type and reduce the subscript range.
- 307    **Description of Error:** Illegal subscript definition.
- Corrective Action:** Check that you have not reversed the order of the subscript delimiters ([3,8] is a valid subscript; [8,3] is invalid) or included a negative number as a subscript delimiter.
- 308    **Description of Error:** Column type not the same for each UNION statement.
- Corrective Action:** Check the select-list of each SELECT statement to make sure that corresponding items have identical data types.
- 309    **Description of Error:** ORDER BY column *column-name* not in SELECT list.
- Corrective Action:** Check that columns included in the ORDER BY clause appear in the SELECT list.
- 310    **Description of Error:** Table *table-name* already exists in database.
- Corrective Action:** Select an alternate name for the table.

- 311    **Description of Error:** Cannot open system catalog *catalog-name*.
- Corrective Action:** Check the ISAM error for information about the source of the problem.
- 312    **Description of Error:** Cannot update system catalog *catalog-name*.
- Corrective Action:** Check the ISAM error for information about the source of the problem.
- 313    **Description of Error:** Not owner of table *table-name*.
- Corrective Action:** Only the owner of the table (or the Database Administrator) can remove the table.
- 314    **Description of Error:** Table *table-name* currently in use.
- Corrective Action:** The table you want to drop is currently being used by another user. Wait until the table is no longer in use before executing your request.
- 315    **Description of Error:** No CREATE INDEX permission.
- Corrective Action:** Permission not granted for you to create an index on the table.
- 316    **Description of Error:** Index *index-name* already exists in database.
- Corrective Action:** An index currently exists for the table. You must drop the existing index before creating a new one.
- 317    **Description of Error:** The same number of selected columns does not appear in each UNION statement.
- Corrective Action:** Check the number of columns selected in each SELECT statement.

- 318    **Description of Error:** File with the same name as specified log file already exists.
- Corrective Action:** Select a different name for the log file.
- 319    **Description of Error:** Index does not exist in database file.
- Corrective Action:** Check the spelling of the index name or check the sysindexes system catalog for the correct index name.
- 320    **Description of Error:** Not owner of index *index-name*.
- Corrective Action:** Only the owner of the index (or the Database Administrator) can remove the index.
- 321    **Description of Error:** Cannot group by aggregate column.
- Corrective Action:** Check the column number used in the GROUP BY clause.
- 322    **Description of Error:** Cannot alter view *view-name*.
- Corrective Action:** Views cannot be altered. You must drop and then recreate the view.
- 323    **Description of Error:** Cannot grant permission on temporary table.
- Corrective Action:** Permissions can only be granted on permanent tables.
- 324    **Description of Error:** Ambiguous column *column-name*.
- Corrective Action:** A column name exists in more than one table cited in your query. Prepend each column name with the appropriate table name.
- 325    **Description of Error:** Full pathname not specified for the log file.
- Corrective Action:** Provide the full pathname for the log file.



- 326    **Description of Error:** Expecting CHAR type host variable.
- Corrective Action:** The cursor is on a CHAR column and your program is attempting to pass it a non-CHAR data type. Restructure your statement.
- 327    **Description of Error:** Cannot unlock table *table-name* within a transaction.
- Corrective Action:** Do not issue UNLOCK TABLE within a transaction.
- 328    **Description of Error:** Column *column-name* already exists in table.
- Corrective Action:** Select a new column name for the column.
- 329    **Description of Error:** Database not found or no system permission.
- Corrective Action:** Check the spelling of the database name. Check that the database name exists in your current directory or a directory included in your DBPATH environment variable. Check the ISAM error for information about the source of the problem.
- 330    **Description of Error:** Cannot create database.
- Corrective Action:** Check that you have not entered the name of an existing database. Select an alternate name for the database. Check the ISAM error for information about the source of the problem.



**-331 Description of Error:** Cannot drop database directory.

**System Action Taken:** All database files in the database directory are deleted, but the directory remains.

**Corrective Action:** Remove any non-database files present in the database directory, then remove the directory. Check the ISAM error for information about the source of the problem.

**-332 Description of Error:** Cannot access audit trail name information.

**Corrective Action:** Re-execute your request. If the error occurs again, the audit trail file is corrupt. You may need to drop and restart the audit trail.

**-333 Description of Error:** The audit trail file already exists with a different name.

**Corrective Action:** You must drop the existing audit trail file (issue a DROP AUDIT FOR statement) before creating a new audit trail.

**-334 Description of Error:** Cannot create audit trail.

**Corrective Action:** You must indicate the full pathname of the file receiving the audit trail. Check that you have permission to write to a file in the selected directory. Contact your System Administrator if you need help with this action.

**-335 Description of Error:** There is no audit trail for the specified table.

**Corrective Action:** INFORMIX-ESQL/C is unable to recover the table as no audit trail was created.

- 336 Description of Error:** Attempt to create or drop audit on temporary table *table-name*.
- Corrective Action:** You cannot place an audit trail on a temporary table.
- 337 Description of Error:** Attempt to create view on temporary table *table-name*.
- Corrective Action:** You cannot create a view on a temporary table.
- 338 Description of Error:** Cannot drop audit trail.
- System Action Taken:** The audit trail was not dropped (possible operating system error.)
- Corrective Action:** Reexecute your request. If the problem re-occurs, check the ISAM error message for information about the source of the problem. Contact your System Administrator if you require assistance with this action.
- 339 Description of Error:** Full pathname not specified for the audit trail file.
- Corrective Action:** Edit your statement to include the full pathname of the audit trail file.
- 340 Description of Error:** Cannot open audit trail file.
- Corrective Action:** Check that you have operating system read permission to the file. Contact your System Administrator if you need help with this action.
- 341 Description of Error:** Cannot read a row from audit trail file.
- Corrective Action:** Reexecute your request. If the error occurs again, the audit trail file is corrupt. You may need to drop and restart the audit trail.

**-343 Description of Error:** Row from audit trail was added to a different position than expected.

**Corrective Action:** Reexecute your request. If the error occurs again, the audit trail file is corrupt. You may need to drop and restart the audit trail.

**-344 Description of Error:** Cannot delete row—row in table does not match row in audit trail.

**Corrective Action:** Reexecute your request. If the error occurs again, the audit trail file is corrupt. You may need to drop and restart the audit trail.

**-345 Description of Error:** Cannot update row—row in table does not match row in audit trail.

**Corrective Action:** Reexecute your request. If the error occurs again, the audit trail file is corrupt. You may need to drop and restart the audit trail.

**-346 Description of Error:** Cannot update a row in the table.

**Corrective Action:** Check the ISAM error for information about the source of the problem.

**-347 Description of Error:** Cannot open table for exclusive access.

**Corrective Action:** Check the ISAM error for information about the source of the problem.

**-348 Description of Error:** Cannot read a row from the table.

**Corrective Action:** Check the ISAM error for information about the source of the problem.



**-349 Description of Error:** Database not selected yet.

**Corrective Action:** Select a database before executing a statement that refers to a database.

**-350 Description of Error:** Index already exists on column.

**Corrective Action:** Adding an index on the column is not necessary; the column is already indexed.

**-351 Description of Error:** Database contains tables owned by other users.

**Corrective Action:** You can only drop a database if you own all tables in the database or have DBA status. Contact your Database Administrator if you need help with this action.

**-352 Description of Error:** Column not found.

**Corrective Action:** Check the spelling of the column name.

**-353 Description of Error:** No table or view specified when granting or revoking table privileges.

**Corrective Action:** You must include the name of the table or view on which a privilege is granted or revoked in your SQL statement.

**-354 Description of Error:** Incorrect database or cursor name format.

**Corrective Action:** A database name must be no longer than 10 characters. A cursor name must be no longer than 18 characters. A database or cursor name must begin with a letter, and contain letters, numbers, or underscores. Check that you have not included an illegal character in the name.

**-355 Description of Error:** Cannot rename file for table.

**Corrective Action:** Check the ISAM error for information about the source of the problem.



**-356 Description of Error:** Table *table-name* specified in both main query and subquery.

**Corrective Action:** The statement is ambiguous because a column cannot be identified uniquely. Use an alias to rename the appropriate table.

**-357 Description of Error:** Dependent table for view *view-name* has been altered.

**Corrective Action:** A table upon which the view is constructed has been modified (for example, a column has been dropped, a data type has been modified, or a column has been added to the middle of the table). Drop the view and create a new view.

**-358 Description of Error:** Database not closed before CREATE, START, or ROLLFORWARD database.

**Corrective Action:** Execute a CLOSE DATABASE statement, then attempt the action.

**-359 Description of Error:** Cannot drop current database.

**Corrective Action:** Execute a CLOSE DATABASE statement before executing a DROP DATABASE statement.

**-360 Description of Error:** Cannot modify table or view used in subquery.

**Corrective Action:** If allowed, your statement could reduce to a looping program. Edit your statement.

**-361 Description of Error:** Column size too large.

**Corrective Action:** Reduce the size of the column. You cannot have a CHAR column larger than 32,511 characters.

**-362 Description of Error:** There is more than one column of SERIAL type.

**Corrective Action:** You cannot have more than one column of SERIAL type in a table. Select an alternate data type for the column.

- 363    **Description of Error:** Cursor not on SELECT statement.

**Corrective Action:** Use EXECUTE to execute prepared objects on non-SELECT statements.

- 364    **Description of Error:** Column *column-name* not declared FOR UPDATE OF.

**Corrective Action:** Include the column in the FOR UPDATE OF list.

- 365    **Description of Error:** Cursor not on simple SELECT for FOR UPDATE.

**Corrective Action:** Check that the cursor does not include more than one table or involve aggregates.

- 366    **Description of Error:** The scale exceeds the maximum precision specified.

**Corrective Action:** The problem is located in a DECIMAL or MONEY column: the scale (number of digits to the right of the decimal point) exceeds the precision (total number of digits).

DECIMAL(m,n) where  $n > m$

MONEY(m,n) where  $n > m$

MONEY(m) where  $m < 2$

- 367    **Description of Error:** Attempt to compute sums or averages for character columns.

**Corrective Action:** Check that you have not included a column of character data type in the aggregate function statement.

- 368    **Description of Error:** Incompatible sqlxec module.

**Corrective Action:** Check that the correct version of **sqlxec** is installed. Contact your Database Administrator if you need help with this action.

- 369 **Description of Error:** Invalid serial number. Consult your installation instructions.

**Corrective Action:** Check that the correct version of **sqlxec** is installed. Contact your Database Administrator if you need help with this action.

- 370 **Description of Error:** Cannot drop last column.

**Corrective Action:** Only one column remains in the table. Use the DROP TABLE statement to remove the table.

- 371 **Description of Error:** The column contains duplicate data.

**Corrective Action:** You cannot create a unique index on a column containing duplicate data.

- 372 **Description of Error:** Attempt to alter table with audit trail on.

**Corrective Action:** You must drop the audit trail before making any changes to the table. After making the changes, you may want to re-establish an audit trail.

- 373 **Description of Error:** DBPATH too long.

**Corrective Action:** Reduce the length of your DBPATH environment variable.

- 374 **Description of Error:** Attempt to use other than a column number in an ORDER BY clause with UNION.

**Corrective Action:** Restructure the query using ordinal numbers for the ORDER BY columns.

- 375 **Description of Error:** Cannot create log file for transaction.



**Corrective Action:** Check the ISAM error for information about the source of the problem.

**-376 Description of Error:** Log file already exists.

**Corrective Action:** Select an alternate name for the log file.

**-377 Description of Error:** Attempt to close database before terminating a transaction.

**Corrective Action:** Issue a COMMIT WORK or ROLLBACK statement before closing the database.

**-378 Description of Error:** Record currently locked by another user.

**Corrective Action:** Wait until the record is unlocked before submitting the statement. Check the ISAM error for information about the source of the problem.

**-379 Description of Error:** Cannot revoke privilege on columns.

**Corrective Action:** Revoke UPDATE or SELECT privilege on the table, then grant the revoked privileges.

**-380 Description of Error:** Cannot erase log file.

**Corrective Action:** Check the ISAM error for information about the source of the problem.

**-381 Description of Error:** Attempt to grant privilege to someone who has granted you the same privilege.

**Corrective Action:** You must remove the name of the individual who granted you permission to use the table from your user list.

**-382 Description of Error:** Same number of columns not specified for view and select clause.



**Corrective Action:** Check the number of columns in the view definition and in the selected columns. Specify the same number for each.

- 383 **Description of Error:** View column for aggregate or expression not explicitly named.

**Corrective Action:** Provide a name for all virtual columns.

- 384 **Description of Error:** Cannot modify non-simple view.

**Corrective Action:** Only modify views built on a single table.

- 385 **Description of Error:** Data value out of range.

**Corrective Action:** Check the view definition for valid data ranges.

- 386 **Description of Error:** Column contains null values.

**Corrective Action:** The table contains NULL values in a column being altered to disallow NULLS. Remove NULL values from the column.

- 387 **Description of Error:** No connect permission.

**Corrective Action:** Contact the Database Administrator and request connect permission.

- 388 **Description of Error:** No resource permission.

**Corrective Action:** Contact the Database Administrator and request resource permission.

- 389    **Description of Error:** No DBA permission.
- Corrective Action:** Contact the Database Administrator and request DBA permission.
- 390    **Description of Error:** Synonym already used as table name or synonym.
- Corrective Action:** Select a different synonym that does not match the name or synonym of any table or view in the current database. Check the **systables** system catalog (where **tabtype** = S) for a list of existing synonyms.
- 391    **Description of Error:** Cannot insert a NULL into column *column-name*.
- Corrective Action:** Check to see if a column that does not allow NULL values has been inserted in the column list.
- 392    **Description of Error:** System error—unexpected null pointer encountered.
- Corrective Action:** Notify the Technical Support Department.
- 393    **Description of Error:** A condition in the where clause results in a two-sided outer join.
- Corrective Action:** A two-sided outer join is not allowed. Restructure your query.
- 394    **Description of Error:** View *view-name* not found.
- Corrective Action:** Check the spelling of the view name. Check the **sysviews** system catalog for a list of existing views.

- 395 Description of Error:** The WHERE clause contains an outer Cartesian product.
- Corrective Action:** Check the syntax of the statement.
- 396 Description of Error:** Illegal join between a nested outer table and a preserved table.
- Corrective Action:** Check the syntax of the statement.
- 397 Description of Error:** System catalog corrupted.
- Corrective Action:** Contact the Database Administrator for help with this error.
- 398 Description of Error:** Cursor manipulation does not fall within a transaction.
- Corrective Action:** Perform a BEGIN WORK statement before any cursor manipulations.
- 399 Description of Error:** Cannot access log file.
- Corrective Action:** You cannot edit the log file.
- 400 Description of Error:** Fetch attempted on unopen cursor.
- Corrective Action:** Check that the cursor was properly opened using an OPEN CURSOR statement.
- 401 Description of Error:** Fetch attempted on NULL cursor.
- Corrective Action:** Check that the cursor was properly opened using an OPEN CURSOR statement.
- 402 Description of Error:** Address of a host variable is NULL.
- Corrective Action:** Check the addresses of each host variable (one or more have a NULL value).



- 403 Description of Error:** The size of a received row disagrees with the expected size.
- Corrective Action:** Check that you are using the proper library in the program.
- 404 Description of Error:** A NULL control block has been passed as an argument.
- Corrective Action:** Check that you are using the proper library in the program.
- 405 Description of Error:** The address of a host variable is not properly aligned.
- Corrective Action:** Check that each host variable is aligned with the proper address boundary for variables of that type.
- 406 Description of Error:** Memory allocation failed.
- Corrective Action:** Exit to the operating system command line, and resubmit your program.
- 407 Description of Error:** Error number zero received from the `sqlexec` process.
- Corrective Action:** Exit to the operating system command line, and resubmit your program.
- 408 Description of Error:** Invalid message type received from the `sqlexec` process.
- Corrective Action:** Exit to the operating system command line, and resubmit your program.
- 409 Description of Error:** `sqlexec` was not found or was not executable by the current user.
- Corrective Action:** Check that your `INFORMIXDIR` environment variable is properly set. Contact your System Administrator if you need help with this action.



**-410 Description of Error:** PREPARE statement failed or was not executed.

**Corrective Action:** Check that your PREPARE statement was successfully executed (a failure is often due to a syntax error).

**-411 Description of Error:** Attempt to specify both host variables and descriptor.

**Corrective Action:** You cannot specify host variables in the DECLARE statement and then use the descriptor option on the OPEN statement (query cursor) or the descriptor option on the PUT statement (insert cursor). Either remove the host variables from the DECLARE statement, or remove references to the descriptor in the OPEN or PUT statement.

**-412 Description of Error:** Command pointer is NULL.

**Corrective Action:** The statement executed prior to the current statement returned an error that was not trapped. Reexecute the prior statement(s) and include a response to the error code returned.

**-413 Description of Error:** Insert attempted on unopened cursor.

**Corrective Action:** Open the cursor before executing a PUT statement.

**-414 Description of Error:** Insert attempted on NULL cursor.

**Corrective Action:** Make sure you have declared the cursor correctly.

**-415 Description of Error:** Data conversion error discovered during PUT operation.

**Corrective Action:** The host variable is incompatible with the data type of a column in the database. Choose an appropriate host variable or restrict the size of the data in the host variable.

- 416 Description of Error:** USING option with OPEN statement is invalid for insert cursor.
- Corrective Action:** You should use the USING option with the PUT statement.
- 417 Description of Error:** FLUSH can only be used on an insert cursor.
- Corrective Action:** Make sure you are using the correct cursor.
- 418 Description of Error:** NULL SQLDA descriptor or host variable list encountered.
- Corrective Action:** Make sure you are using the correct host variable and descriptor.
- 419 Description of Error:** SQLDATA pointer in SQLDA or host variable is null.
- Corrective Action:** Check the host variable list to make sure it is valid. Make sure that memory is properly assigned for the SQLDA structure.
- 461 Description of Error:** File open error.
- Corrective Action:** Verify that the file exists and that it has the correct permissions.
- 462 Description of Error:** File close error.
- Corrective Action:** Verify that the file exists and that it has the correct permissions.
- 463 Description of Error:** File read error.
- Corrective Action:** Verify that the file exists and that it has the correct permissions.

**-464 Description of Error:** File write error.

**Corrective Action:** Verify that the file exists and that it has the correct permissions.

**-465 Description of Error:** No more memory for locator buffer.

**Corrective Action:** Exit to the operating system command line and resubmit your program.

**-500 Description of Error:** Clustered index already exists in the table.

**Corrective Action:** Do not create a clustered index where one already exists. Alter the existing cluster index to NOT CLUSTER before creating a new clustered index.

**-501 Description of Error:** Index is already not clustered.

**Corrective Action:** You do not need to alter index to NOT CLUSTER.

**-502 Description of Error:** Cannot cluster index.

**Corrective Action:** Check the ISAM error code for more information about the problem.

**-503 Description of Error:** Too many tables locked.

**Corrective Action:** Either unlock one or more tables, or open the database IN EXCLUSIVE MODE.

**-504 Description of Error:** Attempt to lock a view.

**Corrective Action:** You cannot lock a view. Edit your program or statements and remove the request to lock the view.



**-505 Description of Error:** Number of columns in UPDATE does not match number of VALUES.

**Corrective Action:** Rewrite your SQL statement.

**-506 Description of Error:** Do not have permission to update all columns.

**Corrective Action:** Obtain permission from the owner of the table.

**-507 Description of Error:** Cursor not found.

**Corrective Action:** This cursor has not been defined. Check the spelling of the cursor.

**-508 Description of Error:** Attempt to rename a temporary file.

**Corrective Action:** You cannot rename a temporary file. Modify your program, or create another table and insert to fill the table.

**-509 Description of Error:** Attempt to rename a column in a temporary file.

**Corrective Action:** Use an alias to define the column name as part of the select statement.

**-513 Description of Error:** Statement not available with this database engine.

**Corrective Action:** Refer to the *INFORMIX-OnLine Programmer's Manual*. Make sure that you are using the appropriate database module (sqlexec for INFORMIX-SE or sqlturbo for INFORMIX-OnLine).

**-515 Description of Error:** Statement not available with this database engine.

**Corrective Action:** Refer to your INFORMIX-ESQL/C manual. Make sure that you are using the appropriate database module (sqlexec for INFORMIX-SE or sqlturbo for INFORMIX-OnLine).



- 528    **Description of Error:** Maximum output rowsize (32767) exceeded.
- Corrective Action:** The total number of bytes for the selected columns is greater than the maximum. You should reduce the number of items in the select list.
- 529    **Description of Error:** Cannot attach to shared memory.
- Corrective Action:** The shared memory may not be up. Check the RSAM error message for the specific reason.
- 531    **Description of Error:** Duplicate column *column-name* exists in view.
- Corrective Action:** Two view columns have the same name; change one of them.
- 532    **Description of Error:** Cannot alter temporary table *table-name*.
- Corrective Action:** A temporary table may not be altered.
- 533    **Description of Error:** Extent size too small, minimum size is %dk.
- Corrective Action:** The default size, 8 kilobytes, is used. The minimum extent size is 8 kilobytes, regardless of page size. Revise the CREATE TABLE statement.
- 534    **Description of Error:** Could not insert new row into table, table is locked.
- Corrective Action:** Wait until table is unlocked and retry the statement.
- 535    **Description of Error:** Already in transaction.
- Corrective Action:** You must issue a COMMIT WORK or ROLLBACK WORK statement at this point.

**-536 Description of Error:** Cannot have more than one cursor on the same statement.

**Corrective Action:** Prepare another identifier on the statement and declare a cursor on that identifier.

**-538 Description of Error:** Cursor *cursor-name* has already been declared.

**Corrective Action:** Check to see where you have declared a cursor with the same name elsewhere and change one of the cursor names.

**-539 Description of Error:** DBTEMP too long.

**Corrective Action:** Shorten the pathname which you have assigned to DBTEMP, or use the default value.

**-541 Description of Error:** User does not have ALTER privilege.

**Corrective Action:** You have not been granted the privilege to alter or rename a table. See the section "User Status and Privileges" in Chapter 1 for appropriate SQL table privilege statements.

**-542 Description of Error:** Cannot specify the same column more than once in a UNIQUE CONSTRAINT.

**Corrective Action:** Delete the redundant reference to the column in the CREATE TABLE or ALTER TABLE statement.

**-549 Description of Error:** Column *column-name* in UNIQUE CONSTRAINT is not a column in the table.

**Corrective Action:** Check the spelling of *column-name*.

**-550 Description of Error:** Total length of columns in UNIQUE CONSTRAINT is too long.

**Corrective Action:** The total length of the composite column list included in a UNIQUE CONSTRAINT cannot exceed 128 bytes. Remove a column from the composite column list.

**-551 Description of Error:** UNIQUE CONSTRAINT contains too many columns.

**Corrective Action:** A composite column list can contain up to 8 column names. Remove a column from the composite column list.

**-559 Description of Error:** Cannot create a synonym on top of another synonym.

**Corrective Action:** You can create a synonym only for a table or view. Check that the synonym name you are trying to use is not the name of an existing synonym. You can look at the systables table to find the names of existing synonyms (where tabtype = S).

**-801 Description of Error:** SQL Edit buffer is full.

**Corrective Action:** You may need to reduce the size of the file, check that it contains ASCII text, or edit it using your system file editor.

**-803 Description of Error:** The file is too large for internal editing.

**Corrective Action:** Check that the file contains ASCII text and reduce the size of the file.

**-905 Description of Error:** Cannot locate `sqlexec tcp` service in `/etc/services`.

**Corrective Action:** Check the `/etc/services` file on the server for the required entries.

**-906 Description of Error:** Cannot locate remote system (check DBPATH).

**Corrective Action:** Check the accuracy of the DBPATH environment variable.



- 907 Description of Error:** Cannot create socket on local system.
- Corrective Action:** Check that TCP/IP is installed and functioning properly throughout the network.
- 908 Description of Error:** Attempt to connect to remote system failed.
- Corrective Action:** Verify that the server machine is up and functioning properly. Check that the **sqlxecd** network server process on the server is running as a background process and was started by root.
- 909 Description of Error:** Invalid database name format.
- Corrective Action:** Check that the requested database name is no longer than 10 characters and contains only letters, numbers, and underscores (\_). The first character must be a letter.
- 910 Description of Error:** Cannot create an **INFORMIX-OnLine** database from an **INFORMIX-SE** client.
- Corrective Action:** Use an **INFORMIX-OnLine** client to create the database.
- 911 Description of Error:** System error - Cannot read from pipe.
- Corrective Action:** Re-enter your request. If the problem persists, refer to your system manual for more information about the source of the problem.
- 912 Description of Error:** Network error - Could not write to remote system.



**Corrective Action:** Re-enter your request. If the problem persists, run your network diagnostics to determine the source of the problem.

- 913 **Description of Error:** Network error - Could not read from remote system.

**Corrective Action:** Re-enter your request. If the problem persists, run your network diagnostics to determine the source of the problem.

- 914 **Description of Error:** System error - Cannot write to pipe.

**Corrective Action:** Re-enter your request. If the problem persists, refer to your system manual for more information about the source of the problem.

- 915 **Description of Error:** Cannot create an INFORMIX-SE database from an INFORMIX-OnLine client.

**Corrective Action:** Use an INFORMIX-SE client to create the database.

- 916 **Description of Error:** Cannot open /etc/mtab.

**Corrective Action:** Make sure that you can read the /etc/mtab file.

- 917 **Description of Error:** Must close current database before using a new database.

**Corrective Action:** Close the current database statement and re-enter your request for a new database.

- 918 **Description of Error:** Unexpected data received from remote system.

**Corrective Action:** Re-enter your request. If the problem persists, run your network diagnostics to determine the source of the problem.

**-919 Description of Error:** System error. Wrong number of arguments to remote **sqlexec** process.

**Corrective Action:** Re-enter your request. If the problem persists, refer to your system manual for more information about the source of the problem.

**-920 Description of Error:** Cannot read host address in network database.

**Corrective Action:** Check the contents of the **/etc/hosts** file.

**-921 Description of Error:** System error. Illegal or wrong number of arguments to **sqlexec** server.

**Corrective Action:** Verify that the versions of **INFORMIX-NET** running on the client and the host are compatible.

**-922 Description of Error:** Cannot get name of current working directory.

**Corrective Action:** Check operating system permissions for the directory and make sure you have permission to access the directory.

**-923 Description of Error:** Informix product licensed for local database access only.

**Corrective Action:** This version of Informix product is not licensed to work with **INFORMIX-NET**. Locate and use the **INFORMIX-NET** version of Informix product.

**-925 Description of Error:** The protocol type should be **tcp**.

**Corrective Action:** Change the protocol type to **tcp** in **\$INFORMIXDIR/etc/sqlhosts**.

**-926 Description of Error:** **INFORMIX-OnLine** is not licensed for distributed data access.

**Corrective Action:** You cannot access tables or databases on remote systems unless you have **INFORMIX-STAR**.

- 927    **Description of Error:** Exceeded limit of maximum number of sites you can reference.
- Corrective Action:** The maximum number of sites that you can reference is 32.
- 928    **Description of Error:** The remote server is not licensed for distributed data access.
- Corrective Action:** You have tried to access a table on an INFORMIX-OnLine system that is not licensed for remote access. You can only access other INFORMIX-STAR systems.
- 930    **Description of Error:** Cannot connect to remote host *hostname*.
- Corrective Action:** Check that the requested hostname is valid. Verify that the requested hostname has an entry in the required files.
- 931    **Description of Error:** Cannot locate *service-name* service/tcp service in */etc/services*.
- Corrective Action:** Check the */etc/services* file on the client for the required entries.
- 932    **Description of Error:** Error on network connection, *system-call* system call failed.
- Corrective Action:** Check that your network hardware and software are installed and functioning properly throughout the network.
- 933    **Description of Error:** Unknown network type specified in DBNETTYPE.
- Corrective Action:** Set your environment variable DBNETTYPE to starlan or tcp/ip.
- 951    **Description of Error:** User is not known on remote host.



**Corrective Action:** Verify that the user has a **login** on the host, and that the network software is properly configured to accept this username. Consult your machine and network user manuals for assistance.

**-952    Description of Error:** User's password is not correct for the remote host.

**Corrective Action:** Check your `/etc/hosts.equiv` file to make sure that it contains the following entries:

- The name of the host machine you are attempting to connect to
- The name of the the client machine you are currently logged into

If one or both of these entries is missing, add the missing name to the `/etc/hosts.equiv` file and re-enter your request.

**-953    Description of Error:** Remote host could not exec `sqlexec` program.

**Corrective Action:** Verify that the `INFORMIXDIR` environment variable was defined on the server when `root` started the `sqlxecd` network server process. If necessary, define `INFORMIXDIR` on the server and restart the `sqlxecd` process in the background.

**-954    Description of Error:** Client is not known to remote host.

**Corrective Action:** Verify that the host machine "knows" the client machine on the network. If you are using TCP/IP, check your `/etc/hosts` file. Consult your network user manual for assistance.

**-955    Description of Error:** Remote host could not receive data from client.

**Corrective Action:** Verify that both the host machine and the client machine are configured properly, and that both machines are aware of each other on the network. Consult your network user manual for assistance.



- 956 **Description of Error:** Client is not in the file  
/etc/hosts.equiv on the remote host.

**Corrective Action:** Verify that the client machine name appears in the /etc/hosts.equiv file on the remote host machine.

- 1200 **Description of Error:** Number is too large for a DECIMAL data type.

**Corrective Action:** You have exceeded the range of numbers allowed as a decimal data type. Allowable decimal numbers range from  $-.1 \times 10^{-128}$  to  $.1 \times 10^{126}$  (with 32 significant digits). Check the size of the number.

- 1201 **Description of Error:** Number is too small for a DECIMAL data type.

**Corrective Action:** You have exceeded the range of numbers allowed as a decimal data type. Allowable decimal numbers range from  $-.1 \times 10^{-128}$  to  $.1 \times 10^{126}$  (with 32 significant digits). Check the size of the number.

- 1202 **Description of Error:** An attempt was made to divide by zero.

**Corrective Action:** Check that you are not attempting to divide a numerical column type by a character column type (for example, 16/Jones) or that the value of the divisor does not equal zero.

- 1203 **Description of Error:** The values used in a MATCH are not both type CHARACTER.

**Corrective Action:** Check that the values included in your MATCH condition are both CHAR types. Use an alternate comparison condition for non-CHAR types.

-1204 **Description of Error:** Invalid year in date.

**Corrective Action:** Acceptable years are 0001 to 9999. If two digits are used, INFORMIX-ESQL/C assumes the year is 19xx. Check the value entered in the date field.

-1205 **Description of Error:** Invalid month in date.

**Corrective Action:** Months can be represented as the number of the month (1 through 12) or the first three letters of the name of the month. Check the value entered in the date field.

-1206 **Description of Error:** Invalid day in date.

**Corrective Action:** Acceptable days are 01 through 31. Check the value entered in the date field.

-1207 **Description of Error:** returned by `rvalstr()`: The resulting string (after conversion) does not fit into the caller-provided buffer. Converted value does not fit into the allotted space.

**Corrective Action:** The converted value contains too many characters to fit into the character buffer passed to `rvalstr()`. Increase the length of the buffer.

-1208 **Description of Error:** returned by `rvaldata()`: The intended destination type (character) is not the same type as the source. There is no conversion from non-character values to character values.

**Corrective Action:** Check that the data types of both columns are the same.

-1209 **Description of Error:** Without any delimiters, the date does not contain exactly six or eight digits.

**Corrective Action:** You must enter either six or eight digits when specifying a data value to represent the date.

- 1210 **Description of Error:** Date could not be converted to month/day/year format.

**Corrective Action:** Dates must be presented as month, day, and year (August 4, 1989 is allowed; 4 August, 1989 is not). Check the sequence of characters entered in the date field.

- 1211 **Description of Error:** Out of memory.

**Corrective Action:** You have exceeded the data space capacity on your machine. Reduce the complexity of your form.

- 1212 **Description of Error:** Date conversion format string does not contain a month, day, and year component.

**Corrective Action:** The FORMAT string used to format a DATE field must contain month, day, and year components. One of these is missing.

- 1213 **Description of Error:** Character to numeric conversion error.

**Corrective Action:** Check that the values in the character string contain only ASCII characters representing numeric data types.

- 1214 **Description of Error:** Value too large to fit in a SMALLINT.

**Corrective Action:** Acceptable SMALLINT values are whole numbers between -32,767 and 32,767. If a larger number is required, you must use the ALTER TABLE statement to modify the column to INTEGER type.

- 1215 **Description of Error:** Value too large to fit in an INTEGER.

**Corrective Action:** Acceptable INTEGER values are whole numbers between -2,147,483,647 and 2,147,483,647. If a larger number is required, you must use the ALTER TABLE statement to modify the column to DECIMAL type.



**-1216 Description of Error:** Illegal exponent.

**Corrective Action:** Check that the exponent is an integer with a value not exceeding 32,767.

**-1217 Description of Error:** The format string is too large.

**Corrective Action:** Reduce the size of the FORMAT string (used to format a DATE field) in the form specification.

**-1218 Description of Error:** String to date conversion error.

**Corrective Action:** Check the format of the DATE data type in the DBDATE environment variable. The format is illegal.

**-1225 Description of Error:** Column does not admit a NULL value.

**Corrective Action:** Check that you did not try to insert a NULL value in a non-NULL column.

**-1226 Description of Error:** Decimal or money value exceeds maximum precision.

**Corrective Action:** Increase the precision of the DECIMAL or MONEY field.

**-1250 Description of Error:** Unable to create pipes.

**Corrective Action:** Check the spelling of the name of the program receiving the output. Check that the program is available on your system. Check that the program exists in a directory accessed in your PATH environment variable. Contact your System Administrator if you need help with these actions.

**-1251 Description of Error:** Unable to create shared memory. semget failed.

**Corrective Action:** This is an internal error. Follow the suggested actions for error -1250. You may also be out of memory. After verifying that the error is not the result of a system limit or problem, please notify the Informix Technical Support Department.



- 1252 **Description of Error:** Unable to create shared memory. shmget failed.

**Corrective Action:** This is an internal error. Follow the suggested actions for error -1250. You may also be out of memory. After verifying that the error is not the result of a system limit or problem, please notify the Informix Technical Support Department.

- 1257 **Description of Error:** Operating system cannot fork process for back end.

**Corrective Action:** This is an internal error. Follow the suggested actions for error -1250. After verifying that the error is not the result of a system limit or problem, please notify the Informix Technical Support Department.

- 1258 **Description of Error:** Cannot attach to shared memory used to communicate with back end.

**Corrective Action:** This is an internal error. Follow the suggested actions for error -1250. After verifying that the error is not the result of a system limit or problem, please notify the Informix Technical Support Department.

- 1260 **Description of Error:** It is not possible to convert between the specified types.

**Corrective Action:** A data type conversion must make sense; some, such as INTERVAL to DATE, or DATETIME to MONEY, are not supported. You may have referenced the wrong variable or column. Make sure that you have specified the data types that you intended, and that any strings representing data values are correctly formatted.

- 1261 **Description of Error:** Too many digits in the first field of datetime or interval.

**Corrective Action:** Specify fewer digits. The default precision is two (2) digits for every field, except *year* (4) and *fraction* (3). You can specify a non-default precision for *fraction* in the range 1 to 5. For the INTERVAL (but not DATETIME) data type, you can specify a non-default precision of up to 9 digits for any field except *fraction*.

- 1262 **Description of Error:** Non-numeric character in datetime or interval.

**Corrective Action:** You can only use digits and the required hyphen ( - ), blank ( ), colon ( : ), and period ( . ) separators in DATETIME and INTERVAL constants, or as the values within literals. See if you used the wrong separator, included an extraneous blank, omitted a digit, omitted a separator, or entered the *name* of a month or day in place of digits.

- 1263 **Description of Error:** A field in a datetime or interval is out of range.

**Corrective Action:** In an INTERVAL field, the absolute value of any field can range from 0 to  $10^n - 1$ , for  $n$  the declared precision. In a DATETIME value, the *year* can range from 1 to 9999, the *month* from 1 to 12, and the *day* from 1 to a maximum from 28 to 31, depending on the specific *month* and *year*. The *hour* must be a positive integer in the range 0 to 23. The *minute* and *second* must be positive integers in the range 0 to 59. The *fraction* can range from 0 to  $10^n - 1$ , for  $n$  the declared precision for the *fraction* field.

- 1264 **Description of Error:** Extra characters at the end of a datetime or interval.

**Corrective Action:** See if you have included a blank within a field, entered too many digits or too many fields, neglected the effect of an EXTEND function, or made a typing mistake.

- 1265 **Description of Error:** Overflow occurred on a datetime or interval operation.

**Corrective Action:** Both DATETIME and INTERVAL values are stored internally as decimals. Your arithmetic operation produced a decimal overflow. Examine your program logic to see if you can change the sequence or precision of your operations to avoid the overflow.

- 1266 **Description of Error:** Intervals or Datetimes are incompatible for the operation.

**Corrective Action:** The arithmetic operations permitted on INTERVAL and DATETIME (and DATE) values are listed in Appendix H. You may have reversed the order of operands, or attempted a meaningless operation, such as adding two DATETIME values. Correct the logic of your program.

- 1267 **Description of Error:** The result of a datetime computation is out of range.

**Corrective Action:** The range of allowed values for DATETIME and INTERVAL fields is described in the suggested actions for error -1263. Some field in your result (probably the *first*) is too large, or is negative, or has an invalid zero value. Check the terms in your calculation and your program logic to see if you can change the sequence, scale, or precision of your operation to avoid the out-of-range results.

- 1268 **Description of Error:** Invalid datetime qualifier.

**Corrective Action:** Check spelling. You are restricted to the keywords YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and FRACTION. Do not append "S" to a keyword.

- 1269 **Description of Error:** Locator conversion error.

**Corrective Action:** This is an internal error. After verifying that the error is not the result of a system limit or problem, please notify the Informix Technical Support Department.

- 2999 **Description of Error:** SQL server terminated.

**Corrective Action:** You may have killed the engine daemon by accidentally killing the wrong process, or an internal error may have overwritten a pipe to the engine. After verifying that the error is not the result of a system limit or problem, please notify the Informix Technical Support Department.



# Index





- .ace extension, meaning of 6-9
- .dat extension, meaning of 1-6
- .dbs extension, meaning of 1-6
- .ec extension, meaning of 2-31
- .idx extension, meaning of 1-6
- .lok extension, meaning of 3-115
- .per extension, meaning of 6-13
- ABSOLUTE keyword, FETCH statement 3-73
- Accessing a database Preface-12
- Access privileges
  - database privileges 3-80
  - for a synonym 3-32
  - GRANT statement 3-79
  - INSERT F-12
  - on views 1-57
  - removing 3-105
  - summary of 1-44
  - table privileges 3-79
  - using UNIX permissions for a database 3-27
- ACEPREP, compiling report specifications with 6-9
- ACE report
  - C program structure 6-13
  - DEFINE section in specification file 6-6
  - example specification file 6-32, 6-34
  - FORMAT section in specification file 6-6
  - general format 6-6
  - how to compile 6-9
  - specification files 6-6
  - using ACEPREP 6-9
- Active set
  - definition of 1-16
  - effect of closing a cursor on 3-20
- ADD keyword, ALTER TABLE statement 3-14
- Aggregate function
  - AVG 3-171
  - COUNT 3-171
  - in a SELECT clause 3-135
  - MAX 3-171
  - MIN 3-172
  - SUM 3-171
  - with DISTINCT 3-172
  - with NULL values 1-50

- ALL keyword
  - GRANT statement 3-80
  - REVOKE statement 3-105
  - SELECT clause 3-134
  - WHERE clause 3-156
- ALTER INDEX statement
  - definition of 1-14
  - syntax and notes 3-12
  - TO CLUSTER option 3-12
- ALTER keyword, GRANT statement 3-79
- ALTER TABLE statement
  - definition of 1-13
  - guidelines for using 3-15
  - owner naming 3-15
  - syntax and notes 3-14
- ANSI
  - and cursor WITH HOLD 3-49
  - and SCROLL cursor 3-49
  - and WHENEVER statement 3-126
  - compatibility with SQL statements 3-7
  - increased functionality of SQL statements 1-5
  - Informix extensions to standard syntax 3-7
  - standards for SQL Preface-15
  - START DATABASE statement 3-116
  - ansi flag 1-25, 1-27, 2-32, 2-33, 2-34, 3-7, 3-49, 3-127, F-37
- ANY keyword, WHERE clause 3-156
- Array
  - and non-NULL SQL value 2-14
  - declaration of variables 2-11
  - host for CHAR 4-10
  - use within ESQL/C statements 2-11
- ASCII files, input data F-9
- ASC keyword, CREATE INDEX statement 3-30
- AS keyword
  - CREATE VIEW statement 3-43
  - GRANT statement 3-80
- Asterisk (\*) notation in a SELECT clause 3-135
- Audit trail
  - compared to transaction 1-36
  - defined 1-35
  - how to create 1-35
  - recovering a table 1-36
  - removing contents of file 1-36

- Audit trail (cont)
  - SQL 3-24, 3-59, 3-99
  - statements that protect table integrity 1-35
  - when to drop 1-36
  - when to use 1-36
- Auto-indexing 1-48
- AVG aggregate function, in a SELECT statement 3-171
- bcheck utility F-4
- BEFORE keyword, ALTER TABLE statement 3-15
- BEGIN WORK statement
  - definition of 1-31
  - syntax and notes 3-18
- BETWEEN keyword, in WHERE clause 3-144
- Blank characters, in input files F-9
- BLOB data Preface-14
- Boolean expression, with NULL values 1-51
- Bourne shell, setting environment variables C-4, Preface-9
- Braces ( { } ) symbols, comment indicator F-12
- bycmpr() library function 5-6
- bycopy() library function 5-8
- byfill() library function 5-9
- byleng() library function 5-10
- BYTE data type Preface-14
- cace
  - and library functions 5-5
  - compiler for C functions called in reports 6-30
- Calling C functions
  - in ACE 6-7
  - in PERFORM 6-10
- CALL keyword
  - in ACE report specification file 6-8
  - in PERFORM form specification file 6-10
- Case sensitivity, and SQL identifiers 1-7
- C data type, correspondence with SQL 4-6
- C function calls
  - DATETIME type routines listed 4-57
  - DATE type routines listed 4-12
  - DECIMAL type routines listed 4-28
  - in ACE reports 6-9
- C function calls (cont)
  - in PERFORM forms 6-13
  - INTERVAL type routines listed 4-57
  - numeric formatting routines listed 4-47
- C functions
  - ACEPREP 6-5
  - calling in form specification file 6-1
  - calling in report specification file 6-7
  - compiling, linking, running 6-30
  - declaring in report specification file 6-6
  - FORMBUILD 6-5
  - in ACE and PERFORM 6-5
  - in ACE control blocks 6-7
  - in expressions 6-11
  - in PERFORM control blocks 6-10
  - to control PERFORM screens 6-20
- CHARACTER data type, with database columns 1-10
- Character position F-11
- CHAR data type
  - acceptable values 3-35
  - CHARACTER synonym 1-10
  - definition of 4-10
  - subscripting columns 3-132
  - with database columns 1-10
- CLOSE DATABASE statement
  - and multi-statement PREPARE 3-22
  - definition of 1-13
  - legal statements following 3-22
  - syntax and notes 3-22
- CLOSE statement
  - and the sqlca record 1-30
  - syntax and notes 3-20
  - with an INSERT cursor 1-28
  - with a SELECT cursor 1-18
- Clustered indexes 1-48
- Clustering
  - ALTER INDEX statement 3-12
  - CREATE INDEX statement 3-29
  - definition of 1-48
- CLUSTER keyword
  - ALTER INDEX statement 3-12
  - CREATE INDEX statement 3-29
- Colon (:)
- as datetime delimiter 4-73, H-7
- as interval delimiter 4-75, H-10



## Colon (:) (cont)

- between main variable and indicator variable 2-15
- delimiter in DATETIME values F-9
- delimiter in INTERVAL values F-9
- field specification separator F-12
- preceding C variables in SQL 1-7
- preceding host variables 1-7, 2-8, 3-131
- preceding label in WHENEVER statement 3-126
- specifying indicator variable 2-15

## Column

- adding 3-14
- changing column values 3-120
- changing data type 3-14
- data 1-10
- DATETIME 3-38
- DATETIME defined 4-71
- DATETIME described H-3
- DATETIME first to last 4-72
- designated as NOT NULL 3-37
- INTERVAL 3-38
- INTERVAL defined 4-73
- INTERVAL described H-8
- INTERVAL first to last 4-74
- modifying 3-14
- naming conventions 1-7, 3-37
- NULL value in 1-49
- removing 3-14
- renaming 3-101
- virtual 1-56, 3-44

## Columns

- in customer table E-4
- in items table E-5
- in manufact table E-6
- in orders table E-4
- in state table E-6
- in stock table E-5
- in system catalogs B-3

Command file, dbload F-10

COMMIT WORK statement

definition of 1-31

syntax and notes 3-23

Compatibility with industry standards  
Preface-15

## Compiler

- and .ec extension 2-31
- cace, for ACE functions 6-30

## Compiler (cont)

- cperf, for PERFORM functions 6-30
- creating an object file 2-31
- diagnosing errors 2-9
- esql, for ESQ/C routines 2-31
- esqlcargs 2-31
- linking library functions 5-5
- preprocessing 2-31
- syntax 2-31

## Compiling

C functions for ACE and PERFORM  
6-30

form specifications 6-13

programs with the ansi flag 3-7

report specifications 6-9

Concurrency control 1-40

CONNECT access privilege 1-44, 3-81

## Constraint

data, on views 1-58

rules for owner naming 1-8

UNIQUE and DROP INDEX  
statement 3-62

## CONSTRAINT keyword

ALTER TABLE statement 3-15

CREATE TABLE statement 3-34

## Control blocks

in ACE specification file 6-7

in PERFORM specification file 6-10

ON BEGINNING 6-12

ON ENDING 6-12

## Conversion

from DATETIME to DATE 4-78

from DATE to DATETIME 4-78

of data types 4-8

when fetching DATETIME 4-76

when fetching INTERVAL 4-76

when storing DATETIME 4-76

when storing INTERVAL 4-76

Converting your databases to the current  
structure F-25

Correspondence between C and SQL 4-6

COUNT aggregate function, in a SELECT  
statement 3-171

## cperf

and library functions 5-5

compiler for C functions called in  
forms 6-30



- C programs
  - compiling for ACE and PERFORM 6-30
  - customizing for ACE and PERFORM 6-13
  - customizing sacego 6-30
  - customizing sperform 6-30
  - linking libraries for ACE and PERFORM 6-30
  - writing for ACE and PERFORM 6-13
- C program structure
  - arg parameter 6-16
  - ctools.h 6-15
  - dynamic SQL statements 2-21
  - embedding SQL statements 2-17
  - error handling 2-18
  - for ACE and PERFORM 6-13
  - header files 2-6
  - host variables 2-8
  - indicator variables 2-14
  - input parameters 6-16
  - return value macros 6-18
  - sqlca structure 2-18
  - sqlda structure 2-21
  - strreturn macro 6-19
  - type conversion functions 6-18
  - userfuncs array 6-15
  - valueptr 6-15
- CREATE AUDIT statement, syntax and notes 3-24
- CREATE DATABASE statement
  - and multi-statement PREPARE 3-27
  - current database 3-26
  - definition of 1-12
  - MODE ANSI keywords 3-26
  - syntax and notes 3-26
  - system catalogs 3-26
  - WITH LOG IN clause 1-32, 3-27
- CREATE INDEX statement
  - definition of 1-14
  - syntax and notes 3-29
  - with ASC keyword 3-30
  - with CLUSTER keyword 3-29
  - with DESC keyword 3-30
  - with UNIQUE or DISTINCT keyword 3-29
- CREATE SYNONYM statement
  - definition of 1-14
  - guidelines for using 3-32
- CREATE SYNONYM statement (cont)
  - owner naming 3-32
  - syntax and notes 3-32
- CREATE TABLE statement
  - DATETIME data type H-5
  - definition of 1-13
  - guidelines for using 3-37
  - Informix extensions to 3-10
  - INTERVAL data type H-11
  - notes 3-37
  - NOT NULL clause 1-49, 3-37
  - owner naming 3-37
  - syntax 3-34
  - TEMP keyword 3-37
  - WITH NO LOG 3-35
- CREATE VIEW statement
  - definition of 1-13, 1-55
  - guidelines for using 3-44
  - owner naming 3-43
  - syntax and notes 3-43
  - WITH CHECK OPTION 1-56, 3-45
- C routines
  - compiling 2-31
  - embedding SQL statements in 2-17
- C shell, setting environment variables
  - C-4, Preface-9
- ctools.h header file
  - and PERFORM 6-15
  - shown A-12
- Currency symbol, in dbload input files F-9
- Current database
  - CLOSE DATABASE statement 3-22
  - closing 3-22
  - CREATE DATABASE statement 3-26
  - creating 3-26
  - DATABASE statement 3-46
  - definition of 1-12
  - DROP DATABASE statement 3-60
  - FETCH statement 3-74
  - how to select 3-46
  - querying 3-110, 3-129
  - SELECT statement 3-110, 3-129
  - UPDATE STATISTICS statement 3-124
- CURRENT function
  - definition of 1-61
  - in a SELECT statement 3-180

## CURRENT keyword

- FETCH statement 3-72
- in SELECT statement 3-145
- replacing DATETIME value H-16
- using for a DATETIME value H-16

## Current row

- definition of 1-16
- deleting 1-20
- updating 1-22

## Cursor

- advancing 1-20, 3-72
- closed state 1-17
- CLOSE statement 3-20
- closing 1-20
- COMMIT WORK statement 3-23
- definition of 1-16
- FOR UPDATE, DELETE statement 3-54
- FOR UPDATE, FETCH statement 3-74
- FOR UPDATE, OPEN statement 3-90
- INSERT, DECLARE statement 3-49
- INSERT, FLUSH statement 3-75
- INSERT, OPEN statement 3-89
- INSERT, PUT statement 3-97
- management 1-16
- naming, with DECLARE 1-28, 3-48
- non-SCROLLing 1-17
- opening 1-19, 3-89
- open state 1-17
- position after DELETE 1-21
- position after UPDATE 1-23
- regular 1-17
- regular, FOR UPDATE 1-17
- SCROLL 1-17
- SCROLL, DECLARE statement 3-49
- SCROLL, FETCH statement 3-73
- SCROLL and MODE ANSI database 1-25
- UPDATE statement 3-121
- WITH HOLD 1-17
- WITH HOLD, CLOSE statement 3-21
- WITH HOLD, DECLARE statement 3-49
- WITH HOLD, OPEN statement 3-90
- WITH HOLD, PUT statement 3-98
- WITH HOLD and locking 1-27
- WITH HOLD and MODE ANSI database 1-27

## Cursor (cont)

- WITH HOLD and transactions 1-25
- WITH HOLD with COMMIT WORK statement 3-23
- with INSERT statement 1-28
- with SELECT statement 3-133
- CURSOR FOR keywords, DECLARE statement 3-48
- CURSOR keyword, DECLARE statement 3-48
- Cursor WITH HOLD
  - and locking 1-27
  - and MODE ANSI database 1-27
  - as regular cursor 1-17
  - as SCROLL cursor 1-17
  - CLOSE statement 3-21
  - COMMIT WORK statement 3-23
  - DECLARE statement 3-49
  - defined 1-25
  - OPEN statement 3-90
  - PUT statement 3-98
- customer table, stores database Preface-11
- Customizing C programs
  - for ACE reports 6-13
  - for PERFORM forms 6-13
- Database
  - access privileges to 1-44
  - closing 3-22
  - committing modifications to 3-23
  - converting to current structure F-25
  - creating 3-26, 3-34
  - creating as MODE ANSI 1-6
  - creating a view for 3-43
  - database subdirectory 1-6, 3-26
  - data manipulation statements 1-15
  - data types for columns 1-10
  - definition of Preface-13
  - deleting an index from 3-62
  - description of system catalogs B-3
  - engine control routine 5-25, 5-26, 5-27
  - engines described Preface-13
  - granting access to 3-79
  - how to open 3-116
  - how to restore a table 3-99
  - map of stores E-7
  - MODE ANSI and ALTER TABLE statement 3-15



## Database (cont)

- MODE ANSI and BEGIN WORK statement 3-18
- MODE ANSI and CREATE DATABASE statement 3-27
- MODE ANSI and CREATE SYNONYM statement 3-32
- MODE ANSI and CREATE TABLE statement 3-37
- MODE ANSI and CREATE VIEW statement 3-43
- MODE ANSI and cursors WITH HOLD 1-27
- MODE ANSI and DELETE statement 3-53
- MODE ANSI and FETCH statement 3-74
- MODE ANSI and INSERT statement 3-85
- MODE ANSI and OPEN statement 3-91
- MODE ANSI and PUT statement 3-98
- MODE ANSI and SCROLL cursors 1-25
- MODE ANSI and START DATABASE statement 3-117
- MODE ANSI and UNLOCK TABLE statement 3-119
- MODE ANSI and UPDATE statement 3-122
- modifying through views 1-56
- naming conventions 1-7, 3-27
- no NULL values for dbupdate utility F-26
- owner naming and MODE ANSI 1-8
- public access Preface-12
- querying current 3-129
- querying through views 1-55
- recovering 1-34
- relational described 1-6
- removing 3-60
- removing a view from 3-67
- restricted access Preface-12
- rolling forward 3-109
- selecting 3-46
- starting 3-116
- stores described E-3
- system catalogs 1-6

## Database (cont)

- tables in 1-6
- transactions 1-31
- two Informix engines Preface-13
- using Demonstration programs Preface-11
- Database engines Preface-13
- DATABASE statement
  - definition of 1-13
  - EXCLUSIVE keyword in 3-46
  - in multi-statement PREPARE 3-47
  - selecting the current database 3-46
  - syntax and notes 3-46
- Database variable, discrepancy with host variable 4-8
- Data conversion
  - among data types 4-8
  - from DATETIME to DATE 4-78
  - from DATE to DATETIME 4-78
  - in an INSERT statement 3-85
  - in an UPDATE statement 3-122
  - when fetching DATETIME 4-76
  - when fetching INTERVAL 4-76
  - when storing DATETIME 4-76
  - when storing INTERVAL 4-76
- Data definition and administration statements 1-12
- Data fields F-9
- Data integrity 1-31
- Data manipulation statements
  - DATETIME H-18
  - DELETE 1-15, 3-53
  - INSERT 1-15, 3-83
  - INTERVAL H-18
  - SELECT 1-15, 3-110, 3-129
  - UPDATE 1-15, 3-120
- Data types
  - and declaration of array 2-11
  - BYTE Preface-14
  - changing 3-14
  - CHAR 1-10, 4-10, F-12
  - CHARACTER 1-10
  - C language 4-6
  - DATE 1-11, 4-12, F-9
  - DATETIME 1-11, 4-70, F-9, H-3
  - DEC 1-10
  - DECIMAL 1-10, 4-26
  - decimal.h 4-27
  - decimal structure 4-27

## Data types (cont)

- `dec_t` 4-27
- defined in `sqltypes.h` 4-6
- DOUBLE 4-12
- DOUBLE PRECISION 1-11
- `dtm_t` 4-70
- `fixchar` 4-11
- FLOAT 1-11, 4-12
- floating decimal point 1-10
- for host variables 4-6
- INT 1-10
- INTEGER 1-10, 4-11
- INTERVAL 1-12, 4-70, F-9, H-8
- `intrvl_t` 4-70
- LONG 4-11
- long 4-11
- MONEY 1-11, 4-26, F-9
- NUMERIC 1-10
  - of columns 1-10
  - of view columns 3-44
- REAL 1-11
- relation of C and SQL 2-10
- SERIAL 1-11, 4-11
- SHORT 4-11
- SMALLFLOAT 1-10, 4-12
- SMALLINT 1-10, 4-11
  - space requirements 3-38
- `sqlda` structure 4-6
- SQL listed 4-6
- string 4-11
- synonyms for 1-10
- TEXT Preface-14
- type conversion 4-8
- VARCHAR Preface-14

## Data validation, using views 1-58

### DATE data type

- acceptable values 3-36
- compatibility with DATETIME H-19
- converting to DATETIME data type 4-78
- definition of 4-12
- format for display fields 3-37
- in `dbload` input files F-9
- `rdatestr()` 4-14
- `rdayofweek()` 4-15
- `rdefmtdate()` 4-16
- representing DATE values H-19
- `rfmtdate()` 4-18
- `rjulmdy()` 4-21

## DATE data type (cont)

- `rleapyear()` 4-22
- `rmdyjul()` 4-23
- `rstrdate()` 4-24
- `rtoday()` 4-25
- with database columns 1-11

### Date function

- DATE 3-174
- DAY 3-175
- in a SELECT statement 3-174
- MDY 3-176
- MONTH 3-177
- WEEKDAY 3-178
- YEAR 3-179

### DATETIME

- columns defined 4-71
- columns described H-3
- converting to DATE 4-78
- data conversion when fetching 4-76
- data conversion when storing 4-76
- data manipulation statements H-18
- declaration of host variable 4-77
- dynamically allocating structures for 2-26
- entering values H-4
- fetching values 4-75
- first to last column 4-72
- manipulating values H-13
- operations using CURRENT keyword H-16
- precision of value adjusted by EXTEND function 3-182
- range of operations for H-12
- storing values 4-76
- value returned by CURRENT function 3-180
- values as character strings H-11
- working with H-3

`datetime.h` header file shown A-15

### DATETIME data type H-3

- acceptable values 3-36
- columns defined 4-71
- compatibility with DATE H-19
- converting to DATE data type 4-78
- data conversion when fetching 4-76
- data conversion when storing 4-76
- declaration 4-77
- definition H-3
- definition of 4-70



## DATETIME data type (cont)

- dcurrent() 4-58
- dctvasc() 4-60
- dtextend() 4-62
- dttoasc() 4-64
- entered as a character string H-11
- entered as a literal value H-4
- fetching values 4-75
- first to last column 4-72
- incvasc() 4-66
- in dbload input files F-9
- intoasc() 4-68
- manipulating arithmetically H-12
- representing DATETIME values H-19
- storing values 4-76
- using the CURRENT keyword H-16
- with database columns 1-11

## Datetime function

- CURRENT 3-180

- EXTEND 3-182

- DAY function, in a SELECT statement 3-175

- DBA access privilege, in GRANT statement 3-79

- DBANSIWARN environment variable 3-7, 3-49, 3-127, C-4

- DBA privilege 1-8, 1-13, 1-14, 1-34, 1-44, 1-58

- DBDATE environment variable C-5

- DBDELIMITER environment variable F-9

- dbexport utility F-33

- dbimport utility F-36

- dbload utility F-8

- DBMONEY environment variable C-6

- DBPATH environment variable C-7

- DBPRINT environment variable C-7

- dbschema utility F-21

- DBTEMP environment variable C-7

- dbupdate utility

- described F-25

- for Version 1.10 database 1-50

- no NULL databases F-26

- decadd() decimal manipulation routine 4-39

- deccmp() decimal manipulation routine 4-41

- deccopy() decimal manipulation routine 4-42

- deccvasc() decimal manipulation routine 4-29

- decdiv() decimal manipulation routine 4-33

- deccvlong() decimal manipulation routine 4-35

- DEC data type, with database columns 1-10

- decdiv() decimal manipulation routine 4-39

- dececv() decimal manipulation routine 4-43

- decfcvt() decimal manipulation routine 4-43

- decimal.h header file shown A-10

## Decimal arithmetic

- addition 4-39

- division 4-39

- multiplication 4-39

- subtraction 4-39

## DECIMAL data type

- acceptable values 3-35

- decadd() 4-39

- deccmp() 4-41

- deccopy() 4-42

- deccvasc() 4-29

- decdiv() 4-33

- deccvlong() 4-35

- decdiv() 4-39

- dececv() 4-43

- decfcvt() 4-43

- decimal structure shown 4-27

- decmul() 4-39

- decround() 4-45

- decsub() 4-39

- DEC synonym 1-10

- dectoasc() 4-31

- dectodbl() 4-38

- dectoint() 4-34

- dectolong() 4-36

- dectrunc() 4-46

- definition of 4-26

- floating decimal point 1-10

- NUMERIC synonym 1-10

- scale and precision 4-9

- with database columns 1-10

## Declaration

- of DATETIME 4-77

- of INTERVAL 4-77

- DECLARE statement
  - and PREPARE statement 3-92
  - definition of 1-28, 1-38
  - FOR UPDATE clause 1-20, 1-22, 3-48
  - guidelines for using 3-49
  - Informix extensions to 3-9
  - syntax and notes 3-48
  - with an INSERT cursor 1-28
  - with a SELECT cursor 1-17
- Declaring C functions in ACE 6-6
- decml() decimal manipulation routine 4-39
- decround() decimal manipulation routine 4-45
- decsb() decimal manipulation routine 4-39
- dectoasc() decimal manipulation routine 4-31
- dectodbl() decimal manipulation routine 4-38
- dectoint() decimal manipulation routine 4-34
- dectolong() decimal manipulation routine 4-36
- dectrunc() decimal manipulation routine 4-46
- dec\_t typedef shown 4-27
- DEFINE section, in ACE report
  - specification file 6-6
- DELETE keyword, GRANT statement 3-79
- DELETE statement
  - definition of 1-15
  - syntax and notes 3-53
  - using DATETIME or INTERVAL values H-18
  - WHERE CURRENT OF clause 1-20, 3-54
- DELIMITER keyword, dbload command file F-10
- demo1 program shown 2-35
- demo2 program shown 2-36
- demo3 program shown 2-38
- Demonstration database
  - creating Preface-11
  - demo1 program 2-35
  - demo2 program 2-36
  - demo3 program 2-38
  - description of Preface-11
- Demonstration database (cont)
  - example programs 2-34
  - installing with esqldemo Preface-11
  - stores, tables in E-3
  - unload program 2-40
- DESC keyword, CREATE INDEX statement 3-30
- DESCRIBE statement
  - syntax and notes 3-56
  - with sqlda structure 2-34
  - with sqlvar\_struct 2-23
  - with sqlwarn 2-20
- DESCRIPTOR keyword
  - in EXECUTE statement 3-68
  - in FETCH statement 3-73
  - in OPEN statement 3-89
  - in PUT statement 3-96
- DISTINCT keyword
  - as synonym for UNIQUE 3-31
  - SELECT clause 3-134
- Distributed query across databases Preface-14
- Dividing INTERVAL values H-17
- Dollar sign (\$)
  - between main variable and indicator variable 2-15
  - default value for DBMONEY C-6
  - displaying a literal 4-52
  - embedding SQL statements in C routines 2-17
  - or EXEC SQL keywords 2-7, 2-17
  - preceding C variables in SQL 1-7
  - preceding host variables 2-8, 3-131
- DOUBLE PRECISION data type, with database columns 1-11
- DROP AUDIT statement
  - definition of 1-35
  - syntax and notes 3-59
- DROP DATABASE statement
  - and multi-statement PREPARE 3-61
  - definition of 1-13
  - syntax and notes 3-60
- DROP INDEX statement
  - and UNIQUE CONSTRAINT 3-62
  - definition of 1-14
  - syntax and notes 3-62
- DROP keyword, ALTER TABLE statement 3-15



- DROP SYNONYM statement
  - definition of 1-14
  - syntax and notes 3-63
- DROP TABLE statement
  - definition of 1-13
  - deleting synonyms 3-65
  - deleting views 3-65
  - syntax and notes 3-65
- DROP VIEW statement
  - definition of 1-14, 1-55
  - syntax and notes 3-67
- dtcurrent() datetime routine 4-58
- dtcvasc() datetime routine 4-60
- dtextend() datetime routine 4-62
- dtime structure shown 4-70
- dtime\_t typedef shown 4-70
- dttoasc() datetime routine 4-64
- Dynamic management statements, sqlda
  - structure 2-21
- Embedding SQL statements 2-17
- Environment variables
  - DBANSIWARN C-4
  - DBDATE C-5
  - DBMONEY C-6
  - DBPATH C-7
  - DBPRINT C-7
  - DBTEMP C-7
  - described C-3
  - how to set C-4, Preface-9
  - INFORMIXDIR C-7, Preface-10
  - PATH Preface-10
  - setting at system prompt Preface-9
  - setting in login file Preface-10
  - setting in profile file Preface-10
  - SQLEXEC C-7
- Error handling 2-18
- Error log file F-8
- Error messages, corrective actions for
  - Error Messages-3
- Error messages, shown in numeric order
  - Error Messages-3
- Errors, trapping with WHENEVER
  - 3-126
- ESCAPE keyword, WHERE clause
  - 3-146, 3-148
- esql
  - and library functions 5-5
  - compiler for ESQ/C 2-31
  - esqldemo, installing demonstration database Preface-11
- EXCLUSIVE keyword
  - DATABASE statement 3-46
  - LOCK TABLE statement 3-87
- Exclusive mode, opening the database
  - 3-117
- EXEC SQL keywords
  - and embedded SQL statements 2-17
  - and host variables 2-7
  - and include file 2-7
- EXECUTE IMMEDIATE statement
  - definition of 1-39
  - syntax and notes 3-70
- EXECUTE statement
  - and PREPARE statement 3-92
  - definition of 1-38
  - DESCRIPTOR keyword in 3-68
  - syntax and notes 3-68
  - USING keyword in 3-68
  - versus DECLARE 3-69
- EXISTS keyword, WHERE clause 3-156
- Expressions
  - Boolean, NULL in 1-51
  - definition of 6-11
  - formatting 4-47
  - in a SELECT statement 3-131
  - NULL values in 1-50
  - PERFORM control block 6-11
- Extend function, in a SELECT statement
  - 3-182
- Extensions to ANSI standard syntax 3-7
- Fetching
  - DATETIME values 4-75
  - INTERVAL values 4-75
- FETCH statement
  - default condition 3-73
  - definition of 1-18
  - guidelines for using 1-18
  - INTO clause 3-73
  - SQLNOTFOUND 3-74
  - syntax and notes 3-72
  - USING DESCRIPTOR clause 3-73
  - with SELECT statement 3-137
- Fields of input records F-9
- File extensions
  - .ace 6-9
  - .dat 1-6
  - .dbs 1-6

## File extensions (cont)

- .ec 2-31
- .idx 1-6
- .lok 3-115
- .per 6-13
- for compiling ACE reports 6-9
- for compiling PERFORM forms 6-13

## FILE keyword

- dbload command file F-10

## Files, header described A-3

## FIRST keyword, FETCH statement 3-72

## Fixed-length records F-9

## FLOAT data type

- acceptable values 3-36
- definition of 4-12
- DOUBLE PRECISION synonym 1-11
- with database columns 1-11

## FLUSH statement

- and the sqlca record 1-30
- guidelines for using 1-28
- syntax and notes 3-75

## FORMAT section, in ACE report

- specification file 6-6

## Formatting numeric expressions

- examples 4-52
- overview 4-47
- valid characters 4-51

## Formatting numeric strings 4-51

## Formatting routines, numeric 4-47

## FORMBUILD

- compiling form specifications with 6-13
- including C functions in form specification 6-5

## Forms, creating with PERFORM 6-10

## FOR TABLE clause, UPDATE STATISTICS statement 3-124

## FOR UPDATE clause, DECLARE statement 3-50

## FOR UPDATE cursor

- DELETE statement 3-54
- FETCH statement 3-74
- OPEN statement 3-90

## FREE statement

- definition of 1-38
- syntax and notes 3-77

## FROM clause

- OUTER keyword 1-59, 3-140
- SELECT statement 3-140

## FROM clause (cont)

- syntax and notes 3-140

## Function

- CURRENT H-16

- UNITS H-16

## FUNCTION keyword, in ACE specification file 6-6

## Function library

- bycmpr() 5-6
- bycopy() 5-8
- byfill() 5-9
- byleng() 5-10
- ldchar() 5-11
- pf\_gettype 6-21
- pf\_getval 6-23
- pf\_msg 6-29
- pf\_nxfield 6-27
- pf\_putval 6-25
- rdownshift() 5-12
- rfmtdec() 4-48
- rfmtdouble() 4-49
- rfmtlong() 4-50
- rgetmsg() 5-13
- risnull() 5-14
- rsetnull() 5-15
- rstod() 5-16
- rstoi() 5-17
- rstol() 5-18
- rtypalign() 5-19
- rtypmsize() 5-20
- rtypname() 5-22
- rtypwidth() 5-23
- rupshift() 5-24
- sqlbreak() 5-25
- sqlexit() 5-26
- sqlstart() 5-27
- stcat() 5-28
- stchar() 5-29
- stcmpr() 5-30
- stcopy() 5-31
- stleng() 5-32
- using cace 5-5
- using cperf 5-5
- using esql 5-5

## Functions

- in SQL statements 3-170
- to control PERFORM screens 6-20

## GRANT statement

- ALL privilege 3-80



## GRANT statement (cont)

- ALTER privilege 3-79
- CONNECT privilege 3-80, 3-81
- DBA privilege 3-81
- definition of 1-14
- DELETE privilege 3-79
- guidelines for using 3-81
- INDEX privilege 3-79
- Informix extensions to 3-10
- INSERT privilege 3-79
- PUBLIC keyword 3-80
- RESOURCE privilege 3-81
- SELECT privilege 3-79
- syntax and notes 3-79
- table privileges 3-80
- UPDATE privilege 3-80
- WITH GRANT OPTION clause 3-80

## GROUP BY clause

- SELECT statement 3-159
- syntax and notes 3-159
- with NULL values 1-53

## HAVING clause

- SELECT statement 3-161
- syntax and notes 3-161

## Header file

- ctools.h 6-15, A-12
- datetime.h A-15
- decimal.h A-10
- sqlca.h 2-6, A-4
- sqllda.h 2-6, A-5
- sqlstype.h 2-6, A-6
- sqltypes.h 2-6, A-8
- syntax for including 2-6

## Host variable

- and standard C typedef expressions 2-10
- as pointer to CHAR 4-10
- C function calls for DATETIME 4-57
- C function calls for INTERVAL 4-57
- converting DECIMAL to FLOAT and DOUBLE 4-12
- data types for 4-6
- declared as C variable 2-8
- declared with C initializer expressions 2-9
- definition of 2-8
- discrepancy with database variable 4-8
- fetching DATETIME value 4-75

## Host variable (cont)

- fetching INTERVAL value 4-75
- in non-parameterized SELECT 2-23
- in parameterized SELECT 2-27
- in SELECT statement 3-131, 3-135, 3-137
- local within a function 2-13
- of type DATETIME 4-77
- of type INTERVAL 4-77
- preceded by \$ 2-8
- preceded by : 2-8
- preceded by EXEC SQL 2-8
- replaced by ? 1-39
- setting to NULL value 2-12
- storing DATETIME value 4-76
- storing INTERVAL value 4-76
- structure of DATETIME value 4-70
- structure of INTERVAL value 4-70
- testing for NULL value 2-12
- warning when re-declaring 2-12

## Hyphen ( - ) symbol

- delimiter in DATETIME values F-9
- delimiter in INTERVAL values F-9
- range indicator in dbload files F-11
- icheck flag, compiling programs with 2-32

## Identifier, SQL 1-7

## Include files 4-70

- datetime.h 4-57
- decimal.h 4-27
- in programs 2-7
- preprocessor statement for 2-7
- sqltypes.h 4-6, 4-7
- syntax for 2-6
- to check success of ESQ/C statements 2-6

## incvasc() datetime routine 4-66

## Index

- altering 3-12
- auto- 1-48
- clustered 1-48
- creating 3-29
- deleting when drop database 3-60
- NULL value in 1-49
- removing from a database 3-62
- removing when drop table 3-65
- use of SET EXPLAIN statement 1-47
- when to index a table 1-45
- with dbload utility F-19

INDEX keyword, GRANT statement 3-79  
 INDICATOR keyword  
     and indicator variable 2-15  
     in SELECT statement 3-138  
 Indicator variable  
     and associated host variable 2-14  
     and INDICATOR keyword 2-15  
     definition of 2-14  
     how to specify in SQL statement 2-15  
     in EXECUTE statement 3-69  
     in INSERT statement 3-85  
     in OPEN statement 3-90  
     main variable 2-14  
     specification of 2-15  
     truncation of 2-14  
     with NULL and NOT NULL values 2-14  
 Industry standards, compatibility with Preface-15  
 INFORMIX-OnLine database engine  
     and BLOBs Preface-14  
     and O symbol in syntax Preface-9  
     and SQLEXEC C-7  
     and VARCHARs Preface-14  
     described Preface-13  
     importing to F-37  
     programming issues using Preface-15  
     SQL statements supporting 3-7  
     statements not supported 3-6  
 INFORMIX-SE database engine  
     and nonshared memory Preface-13  
     and shared memory Preface-13  
     and SQLEXEC C-7  
     described Preface-13  
     importing to F-37  
     SQL statements supported 3-6  
 INFORMIXDIR, how to set Preface-10  
 INFORMIXDIR environment variable C-7  
 IN keyword, WHERE clause 3-156  
 Input files, dbload utility F-9  
 Input parameters, in C program structure 6-16  
 INSERT cursor  
     and FLUSH statement 3-75  
     defined 1-16  
     guidelines for using 1-28  
     in CLOSE statement 3-20  
     OPEN statement 3-89

INSERT cursor (cont)  
     PUT statement 3-97  
 INSERT INTO keywords, INSERT statement 3-83  
 INSERT keyword  
     dbload command file F-10  
     GRANT statement F-12  
 INSERT keyword, GRANT statement 3-79  
 INSERT statement  
     definition of 1-15  
     guidelines for using 3-84  
     inserting through a view 1-56  
     SERIAL columns in 3-84  
     syntax and notes 3-83  
     using DATETIME or INTERVAL values H-11  
     with NULL values 1-53  
 INT data type, with database columns 1-10  
 INTEGER data type  
     acceptable values 3-35  
     definition of 4-11  
     INT synonym 1-10  
     with database columns 1-10  
 Interactive mode, dbload utility F-17  
 Interrupt key F-19  
 INTERVAL  
     columns defined 4-73  
     columns described H-8  
     data conversion when fetching 4-76  
     data conversion when storing 4-76  
     data manipulation statements H-18  
     declaration of host variable 4-77  
     dynamically allocating structures for 2-26  
     entering values H-9  
     fetching values 4-75  
     first to last column 4-74  
     manipulating values H-15  
     multiplying or dividing values H-17  
     operations using UNITS keyword H-16  
     range of operations for H-12  
     storing values 4-76  
     values as character strings H-11  
     working with H-3  
 INTERVAL data type H-3  
     acceptable values 3-36

## INTERVAL data type (cont)

- columns defined 4-73
- data conversion when fetching 4-76
- data conversion when storing 4-76
- declaration 4-77
- definition H-8
- definition of 4-70
- entered as a character string H-11
- entered as a literal value H-9
- fetching values 4-75
- first to last column 4-74
- in dbload input files F-9
- manipulating arithmetically H-12
- storing values 4-76
- using the UNITS keyword H-16
- with database columns 1-12

intoasc() datetime routine 4-68

## INTO clause

- INDICATOR keyword 3-138
- syntax and notes 3-137

## INTO keyword

- dbload command file F-11
- FETCH statement 3-73
- SELECT statement 3-137

## INTO TEMP clause

- SELECT statement 3-165
- syntax and notes 3-165

intrvl structure shown 4-70

intrvl\_t typedef shown 4-70

## IS NULL keywords

- in NULL and WHERE clause 1-52
- in WHERE clause 3-150

items table, stores database Preface-11

## Join

- in a SELECT statement 3-152
- joined columns with the same name 3-153
- multiple joins 3-153
- outer described G-3
- outer join 1-59, 3-153
- self-join 3-153
- where a column is NULL 1-52

## Joins E-8

LAST keyword, FETCH statement 3-72

ldchar() library function 5-11

## LENGTH keyword

- data manipulation statements 3-173
- in a SELECT statement 3-173

## Library functions

- bycmpr() 5-6
- bycopy() 5-8
- byfill() 5-9
- byleng() 5-10
- in PERFORM 6-20
- ldchar() 5-11
- pf\_gettype() 6-21
- pf\_getval() 6-23
- pf\_msg() 6-29
- pf\_nxfield() 6-27
- pf\_putval() 6-25
- rdownshift() 5-12
- rfmtdec() 4-48
- rfmtdouble() 4-49
- rfmtlong() 4-50
- rgetmsg() 5-13
- risnull() 5-14
- rsetnull() 5-15
- rstod() 5-16
- rstoi() 5-17
- rstol() 5-18
- rtypalign() 5-19
- rtypmsize() 5-20
- rtypname() 5-22
- rtypwidth() 5-23
- rupshift() 5-24
- sqlbreak() 5-25
- sqlexit() 5-26
- sqlstart() 5-27
- stcat() 5-28
- stchar() 5-29
- stcmpr() 5-30
- stcopy() 5-31
- stleng() 5-32

LIKE keyword, WHERE clause 3-146

Linking C functions for ACE and  
PERFORM 6-30

## Locking

- and cursors WITH HOLD 1-27
- and the BEGIN WORK statement 3-18

row-level 1-41

table-level 1-42, F-18

waiting for locked row 1-43

Locking statements 1-37

LOCK TABLE statement

definition of 1-37

in EXCLUSIVE MODE 3-87



## LOCK TABLE statement (cont)

in SHARE MODE 3-87

syntax and notes 3-87

## Logging

and MODE ANSI database 1-33, F-37

buffered or unbuffered F-37

choices with INFORMIX-OnLine

Preface-14

creating transaction log file 1-32

database without transactions 1-33

database with transactions 1-33

error messages F-18

maintaining transaction log file 1-34

of imported database F-36, F-37

of temporary tables 1-32, 3-35, 3-165

preventing on temporary tables 1-32, 3-165

versus audit trails 1-35, 1-36

when checking database for errors 1-34

login file, setting environment variables in Preface-10

Managing cursors 1-16

Manipulating DATETIME values H-13

Manipulating INTERVAL values H-15

manufact table, stores database

Preface-11

MATCHES keyword, WHERE clause 3-148

MAX aggregate function, in a SELECT statement 3-171

Mdy function, in a SELECT statement 3-176

Memory, shared and nonshared on INFORMIX-SE Preface-13

Messages, corrective action for errors Error Messages-3

Messages, meaning of errors Error Messages-3

MIN aggregate function, in a SELECT statement 3-172

Minus ( - ) sign

delimiter in DATETIME values F-9

delimiter in INTERVAL values F-9

MODE ANSI database

checking for compatibility 3-7

CREATE TABLE statement 3-37

MODE ANSI keywords

CREATE DATABASE statement 3-26

## MODE ANSI keywords (cont)

START DATABASE statement 3-116

MODE keyword, LOCK TABLE statement 3-87

MODIFY keyword, ALTER TABLE statement 3-15

MONEY data type

acceptable values 3-36

definition of 4-26

in dbload input files F-9

with database columns 1-11

Month function, in a SELECT statement 3-177

Multiplying INTERVAL values H-17

Naming an object in a MODE ANSI database 1-8

Naming conventions

columns 1-7, 3-37

database 1-7, 3-27

table 1-7, 3-37

NEWLINE character F-9

NEXT keyword, FETCH statement 3-72

Non-parameterized SELECT statements 2-23

Notational conventions used in manual Preface-7

NOT NULL keywords, dbload utility F-12

NULL keyword, dbload command file F-11

NULL values

avoiding in dbupdate utility F-26

definition of 1-49

in a column 1-49

in a GROUP BY clause 1-53

in an ORDER BY clause 1-52

in Boolean expressions 1-51

in expressions 1-50

in joined columns 1-52

in WHERE clause 1-51

NOT NULL clause 1-49, 3-37

truth tables 1-51

with dbload utility F-9

with INSERT 1-53

with UPDATE 1-53

NUMERIC data type, with database columns 1-10

Numeric expressions

example formats 4-52



- Numeric expressions (cont)
  - formatting 4-47
  - rfmtdec() routine 4-48
  - rfmtdouble() routine 4-49
  - rfmtlong() routine 4-50
  - valid characters 4-51
- Numeric formatting routines 4-47
- Object, naming in MODE ANSI database 1-8
- ON BEGINNING, PERFORM control blocks 6-12
- ON ENDING, PERFORM control blocks 6-12
- Opening a database 3-116
- OPEN statement
  - and PREPARE statement 3-92
  - definition of 1-28
  - syntax and notes 3-89
  - with an INSERT cursor 1-28
  - with a SELECT cursor 1-18
- Operator, UNION 3-167
- ORDER BY clause
  - ASC keyword in 3-163
  - DESC keyword in 3-163
  - in an INSERT statement 3-84
  - SELECT statement 3-163
  - syntax and notes 3-163
  - with NULL values 1-52
- orders table, stores database Preface-11
- O symbol in syntax descriptions Preface-9
- Outer joins
  - described G-3
  - in SELECT statement 3-153
  - OUTER keyword in SELECT statement 3-140
  - overview 1-59
- OUTER keyword, in FROM clause 3-140
- Owner naming
  - ALTER TABLE statement 3-15
  - and MODE ANSI database 1-8
  - and system catalogs 1-9
  - CREATE SYNONYM statement 3-32
  - CREATE TABLE statement 3-37
  - CREATE VIEW statement 3-43
  - START DATABASE statement 3-117
- Parameterized non-SELECT statements 2-29
- Parameterized SELECT statements 2-27

- PATH, how to set Preface-10
- PERFORM
  - CALL keyword 6-10
  - C functions 6-10
  - C program structure 6-13
  - example specification file 6-36, 6-39
  - expression 6-11
  - form specification files 6-10
  - general format 6-10
  - how to compile 6-13
  - INSTRUCTIONS section 6-10
  - library functions 6-20
  - ON BEGINNING control block 6-12
  - ON ENDING control block 6-12
- PERFORM functions
  - pf\_gettype() 6-21
  - pf\_getval() 6-23
  - pf\_msg() 6-29
  - pf\_nxfield() 6-27
  - pf\_putval() 6-25
- Period ( . ) symbol
  - delimiter in DATETIME values F-9
  - delimiter in INTERVAL values F-9
- pf\_gettype() function in PERFORM 6-21
- pf\_getval() function in PERFORM 6-23
- pf\_msg() function in PERFORM 6-29
- pf\_nxfield() function in PERFORM 6-27
- pf\_putval() function in PERFORM 6-25
- PREPARE statement
  - and multi-statement string 3-93
  - definition of 1-38
  - executing 3-68
  - executing immediately 3-70
  - guidelines for using 3-92
  - syntax and notes 3-92
- Preprocessor
  - \$define statement 2-29
  - \$else statement 2-30
  - \$endif statement 2-30
  - \$ifdef statement 2-30
  - \$ifndef statement 2-30
  - \$include statement 2-7, 2-29
  - \$undef statement 2-29
  - conditional compilation of ESQL/C statements 2-29
  - converting embedded SQL into C source code 2-29
  - EXEC SQL include statement 2-7

## Preprocessor (cont)

- search sequence for included files 2-30

- statements handled 2-29

PREVIOUS keyword, FETCH statement 3-72

PRIOR keyword, FETCH statement 3-72

## Privileges

- database access 1-44

- removing 3-105

## PRIVILEGES keyword

- GRANT statement 3-80

- REVOKE statement 3-105

profile file, setting environment variables in Preface-10

## Program

- ACEPREP 6-5

- compiling 2-31

- default print C-7

- demonstration database samples 6-31

- example, a\_ex1.ace 6-32

- example, a\_ex2.ace 6-34

- example, decsqrt.c 6-34

- example, demo1 2-35

- example, demo2 2-36

- example, demo3 2-38

- example, mult\_item.ec 6-44

- example, p\_ex1.per 6-36

- example, p\_ex2.per 6-44

- example, stamp.c 6-38

- example, to\_unix.c 6-32

- example, unload 2-40

- FORMBUILD 6-5

- including header files 2-6

- preprocessing 2-31

- preprocessor statement to include files 2-7

- utility, bcheck F-4

- utility, dbexport F-33

- utility, dbimport F-36

- utility, dbload F-8

- utility, dbschema F-21

- utility, dbupdate F-25

- utility, sqlconv F-27

- writing C for ACE and PERFORM 6-13

Programming with INFORMIX-OnLine database engine Preface-15

Prompt, setting environment variables at Preface-9

## PUBLIC keyword

- GRANT statement 3-80

- REVOKE statement 3-106

## PUT statement

- and the sqlca record 1-30

- guidelines for using 1-28

- syntax and notes 3-96

## Querying the database

- compound queries 3-167

- distributed across databases

- Preface-14

- opening a SELECT cursor 3-89

- searching for rows with NULL values 1-52

- subqueries 3-121, 3-155

- through views 1-55

## Query optimizer 1-47

Range of values, dbload utility F-12

rdatestr() date manipulation routine 4-14

rdayofweek() date manipulation routine 4-15

rdefmtdate() date manipulation routine 4-16

rdownshift() library function 5-12

REAL data type, with database columns 1-11

Records in data files F-9

RECOVER TABLE statement, syntax and notes 3-99

RELATIVE keyword, FETCH statement 3-72

## RENAME COLUMN statement

- definition of 1-14

- syntax and notes 3-101

## RENAME TABLE statement

- definition of 1-13

- syntax and notes 3-103

Reports, creating with ACE 6-6

Reserved words, listing D-3

RESOURCE access privilege 1-44, 3-81

Restoring a database table 3-99

Restricting access to a database Preface-12

Return value macros, in C program structure 6-18



## REVOKE statement

- ALL privilege 3-105
- ALTER privilege 3-105
- CONNECT privilege 3-106
- DBA privilege 3-106
- definition of 1-14
- DELETE privilege 3-105
- guidelines for using 3-106
- INDEX privilege 3-105
- INSERT privilege 3-105
- PUBLIC keyword 3-106
- RESOURCE privilege 3-106
- SELECT privilege 3-105
- syntax and notes 3-105
- UPDATE privilege 3-105

rfmtdatetime() date manipulation routine  
4-18

rfmtdec() numeric formatting routine  
4-48

rfmtdouble() numeric formatting routine  
4-49

rfmtlong() numeric formatting routine  
4-50

rgetmsg() library function 5-13

risnull() library function 5-14

rjulmdy() date manipulation routine  
4-21

rleapyear() date manipulation routine  
4-22

rmidyjul() date manipulation routine  
4-23

## ROLLBACK WORK statement

- definition of 1-31
- syntax and notes 3-108

## ROLLFORWARD DATABASE statement

- definition of 1-32
- recovering a database 1-34
- syntax and notes 3-109

## Routine

- decadd() 4-39
- deccmp() 4-41
- deccopy() 4-42
- deccvasc() 4-29
- deccvdbl() 4-37
- deccvint() 4-33
- deccvlong() 4-35
- decdiv() 4-39
- dececv() 4-43
- decfcvt() 4-43

## Routine (cont)

- decmul() 4-39
- decround() 4-45
- decsub() 4-39
- dectoasc() 4-31
- dectodbl() 4-38
- dectoint() 4-34
- dectolong() 4-36
- dectrunc() 4-46
- dtcurrent() 4-58
- dtcvasc() 4-60
- dttextend() 4-62
- dttoasc() 4-64
- incvasc() 4-66
- intoasc() 4-68
- rdatestr() 4-14
- rdayofweek() 4-15
- rdefmtdatetime() 4-16
- rfmtdatetime() 4-18
- rfmtdec() 4-48
- rfmtdouble() 4-49
- rfmtlong() 4-50
- rjulmdy() 4-21
- rleapyear() 4-22
- rmidyjul() 4-23
- rstrdate() 4-24
- rtoday() 4-25

## Row

- active set of rows 1-16
- current row 1-16
- definition of 1-6
- deleting current 1-20
- deleting from a table 3-53
- duplicates in a view 1-57
- ROWID for table access 1-60
- updating current 1-22
- waiting for locked 1-43

rsetnull() library function 5-15

rstod() library function 5-16

rstoi() library function 5-17

rstol() library function 5-18

rstrdate() date manipulation routine  
4-24

rtoday() date manipulation routine 4-25

rtypalign() library function 5-19

rtypmsize() library function 5-20

rtypname() library function 5-22

rtypwidth() library function 5-23

Running C functions for ACE and

PERFORM 6-30

rupshift() library function 5-24

sacego, customizing for ACE 6-30

saceprep, use with ACE 6-9

SCROLL cursor

and MODE ANSI database 1-25

DECLARE statement 3-49

fetching rows into temporary file 1-23

relationship to FETCH statement

3-73

SCROLL keyword, DECLARE statement

3-48

SELECT clause

ALL keyword 3-134

and multi-statement PREPARE 3-135

display label 3-135

DISTINCT keyword 3-134

guidelines for using 3-134

syntax and notes 3-134

UNIQUE keyword 3-134

SELECT cursor

defined 1-16

with CLOSE statement 1-18

with OPEN statement 1-18

SELECT keyword

dbload command file F-10

SELECT keyword, GRANT statement

3-79

SELECT statement

active rows returned 1-16

aggregate functions in 3-171

and PREPARE statement 3-92

clauses shown 3-110

creating temporary tables 3-165

current row in 1-16

date functions in 3-174, 3-175, 3-176,

3-177, 3-178, 3-179

datetime functions in 3-180, 3-182

defined 1-15, 3-129

defining a view 1-55

determining resources used by 3-111

distributed across databases

Preface-14

expression in 3-131

FROM clause 3-140

functions in 3-170

GROUP BY clause 3-159

HAVING clause 3-161

SELECT statement (cont)

Informix extensions to 3-8

INSERT statement 3-83

INTO clause 3-73, 3-137

INTO TEMP clause 3-165

joining columns with 3-152

LENGTH keyword in 3-173

non-parameterized 2-23

notes 3-132

ORDER BY clause 3-163

overview of clauses 3-132

parameterized 2-27

relational operators 3-132

SELECT clause 3-134

singleton SELECT statements 3-137

syntax 3-129

UNION operator 3-167

use of INDICATOR keyword 3-138

using DATETIME or INTERVAL

values H-18

WHERE clause 3-142

with an outer join 1-59, 3-153

with a self-join 3-153

with multiple joins 3-153

WITH NO LOG option 3-165

with subqueries 3-155

Semicolon (;) symbol, statement

terminator F-12

SERIAL data type

acceptable values 3-36

definition of 4-11

INSERT statement 3-84

with database columns 1-11

SET EXPLAIN statement, syntax and

notes 3-111

SET keyword, UPDATE statement 3-120

SET LOCK MODE statement

and kernel locking 3-115

syntax and notes 3-114

WAIT keyword 3-114

Setting environment variables C-4,

Preface-9

sformbld, use with PERFORM 6-13

SHARE keyword, LOCK TABLE

statement 3-87

SMALLFLOAT data type

acceptable values 3-36

definition of 4-12

REAL synonym 1-11



**SMALLFLOAT data type (cont)**

with database columns 1-10

**SMALLINT data type**

acceptable values 3-35

definition of 4-11

with database columns 1-10

**SOME keyword, WHERE clause 3-156**

**Sorting data**

DATE columns 3-38

indexing strategy 1-45

nested sort 3-164

when to create temporary table 1-46

with NULL values 1-52

with ORDER BY clause 1-52, 3-163

**Specification file**

ACE 6-5

ACE example 6-32, 6-34

PERFORM 6-5

PERFORM example 6-36, 6-39

sperform, customizing for PERFORM  
6-30

**SQL**

ALTER INDEX statement 3-12

ALTER TABLE statement 3-14

and ANSI standards Preface-15

and EXEC SQL keywords 2-17

audit trails 3-24, 3-59, 3-99

BEGIN WORK statement 3-18

CLOSE DATABASE statement 3-22

CLOSE statement 3-20

CLOSE statement and cursors 1-18

CLOSE statement with INSERT

cursor 1-28

COMMIT WORK statement 3-23

correspondence to C data types 4-6

CREATE AUDIT statement 3-24, 3-99

CREATE DATABASE statement 3-26

CREATE INDEX statement 3-29

CREATE SYNONYM statement 3-32

CREATE TABLE statement 3-34

CREATE VIEW statement 3-43

database administration statements

1-12, 1-14

DATABASE statement 3-46

data conversion 3-85

data definition statements 1-12

data integrity statements 1-31

data manipulation statements 1-15

DECLARE statement 3-48

**SQL (cont)**

DECLARE statement for cursor  
management 1-17

DECLARE statement with INSERT  
cursor 1-28

DELETE statement 3-53

DESCRIBE statement 3-56

DROP AUDIT statement 3-59, 3-99

DROP DATABASE statement 3-60

DROP INDEX statement 3-62

DROP SYNONYM statement 3-63

DROP TABLE statement 3-65

DROP VIEW statement 3-67

dynamic management statements  
1-38

dynamic statements and sqlda 2-21  
embedding statements in C routines  
2-17

EXECUTE IMMEDIATE statement  
3-70

EXECUTE statement 3-68

FETCH statement 3-72

FETCH statement and cursors 1-18

FLUSH statement 3-75

FLUSH statement with INSERT  
cursor 1-28

FREE statement 3-77

functions in statements 3-170

GRANT statement 3-79

identifiers 1-7

incorporating Version 1.10 database  
1-50

Informix extensions to 1-5

INSERT statement 3-83

interactive query language 1-5

list of statements 3-5

LOCK TABLE statement 3-87

OPEN statement 3-89

OPEN statement and SELECT cursor  
1-18

OPEN statement with INSERT cursor  
1-28

PREPARE statement 3-92

PUT statement 3-96

PUT statement with INSERT cursor  
1-28

RECOVER TABLE statement 3-99

RENAME COLUMN statement 3-101

RENAME TABLE statement 3-103

## SQL (cont)

- REVOKE statement 3-105
- ROLLBACK WORK statement 3-108
- ROLLFORWARD DATABASE
  - statement 3-109
- SELECT statement 3-110, 3-129
- SET EXPLAIN statement 3-111
- SET LOCK MODE statement 3-114
- START DATABASE statement 3-116
  - statements supported only on INFORMIX-SE 3-6
  - statements that support INFORMIX-OnLine 3-7
- subscripting CHAR columns 3-132
- syntax conventions Preface-7
- UNLOAD statement F-9
- UNLOCK TABLE statement 3-119
- UPDATE statement 3-120
- UPDATE STATISTICS statement 3-124
- WHENEVER statement 3-126
- sqlbreak() library function 5-25
- sqlca.h header file shown A-4
- sqlca.sqlcode, DESCRIBE statement 3-56
- sqlca record
  - and CLOSE statement 1-30
  - and FLUSH statement 1-30
  - and PUT statement 1-30
- sqlca structure
  - and error handling 2-18
  - and truncation 2-14
  - definition of 2-18
- sqlcode defined 2-18
- sqlconv utility
  - description of F-27
  - no shortage of disk space for F-28
  - shortage of disk space for F-29
- sqlda.h header file shown A-5
- sqlda structure
  - data types 4-6
  - definition of 2-21
  - DESCRIBE statement 3-56
  - dynamic SQL statements 2-21
- sqlerrd defined 2-19
- SQLERROR keyword, in WHENEVER statement 3-126
- SQLEXEC environment variable C-7
- sqlexit() library function 5-26

SQLNOTFOUND, FETCH statement 3-74

sqlstart() library function 5-27  
sqlstype.h header file shown A-6  
sqltypes.h header file shown A-8  
sqlwarn

- sqlwarn0 described 2-19
- sqlwarn1 described 2-19
- sqlwarn2 described 2-19
- sqlwarn3 described 2-20
- sqlwarn4 described 2-20
- sqlwarn5 described 2-20
- structure 2-19

SQLWARNING keyword, in WHENEVER statement 3-126

START DATABASE statement

- definition of 1-32
- MODE ANSI keywords 3-116
- syntax and notes 3-116
- uses 3-116

## Statements

- DATETIME data manipulation H-18
- functions in SQL 3-170
- Informix extensions to ANSI standard syntax 3-7
- INTERVAL data manipulation H-18
- SQL list 3-5
- supporting INFORMIX-OnLine enhancements 3-7
- syntax conventions for Preface-7

## Statement syntax

- ALTER INDEX 3-12
- ALTER TABLE 3-14
- BEGIN WORK 3-18
- CLOSE 3-20
- CLOSE DATABASE 3-22
- COMMIT WORK 3-23
- CREATE AUDIT 3-24
- CREATE DATABASE 3-26
- CREATE INDEX 3-29
- CREATE SYNONYM 3-32
- CREATE TABLE 3-34
- CREATE VIEW 3-43
- DATABASE 3-46
- DECLARE 3-48
- DELETE 3-53
- DESCRIBE 3-56
- DROP AUDIT 3-59
- DROP DATABASE 3-60



## Statement syntax (cont)

- DROP INDEX 3-62
- DROP SYNONYM 3-63
- DROP TABLE 3-65
- DROP VIEW 3-67
- EXECUTE 3-68
- EXECUTE IMMEDIATE 3-70
- FETCH 3-72
- FLUSH 3-75
- FREE 3-77
- GRANT 3-79
- INSERT 3-83
- LOCK TABLE 3-87
- OPEN 3-89
- PREPARE 3-92
- PUT 3-96
- RECOVER TABLE 3-99
- RENAME COLUMN 3-101
- RENAME TABLE 3-103
- REVOKE 3-105
- ROLLBACK WORK 3-108
- ROLLFORWARD DATABASE 3-109
- SELECT 3-129
- SET EXPLAIN 3-111
- SET LOCK MODE 3-114
- START DATABASE 3-116
- UNLOCK TABLE 3-119
- UPDATE 3-120
- UPDATE STATISTICS 3-124
- WHENEVER 3-126

## Statement type

- database administration 1-12
- data definition 1-12
- data integrity 1-31
- data manipulation 1-15
- dynamic management 1-38
- state table, stores database Preface-11
- Status, user 1-44
- stcat() library function 5-28
- stchar() library function 5-29
- stcmpr() library function 5-30
- stcopy() library function 5-31
- stleng() Library function 5-32
- stock table, stores database Preface-11
- stores database
  - customer table Preface-11
  - customer table columns E-4
  - data values E-14
  - described E-3

## stores database (cont)

- items table Preface-11
- items table columns E-5
- join columns E-8
- manufact table Preface-11
- manufact table columns E-6
- orders table Preface-11
- orders table columns E-4
- state table Preface-11
- state table columns E-6
- stock table Preface-11
- stock table columns E-5
- structure of tables E-4
- tables in Preface-11

## Storing DATETIME values 4-76

## Storing INTERVAL values 4-76

## Strings, formatting numeric 4-51

## struct

- decimal shown 4-27
- dtime shown 4-70
- intrvl shown 4-70
- sqlca shown 2-18
- sqlvar shown 2-21

## Structures

- declared as host objects 2-9
- dynamic SQL statements and sqllda 2-21
- error handling and sqlca 2-18
- nesting 2-9
- sqllda defined 2-21
- using sqllda 2-28

## SUM aggregate function, in a SELECT statement 3-171

## Synonym

- creating 3-32
- deleting when remove table 3-65
- for a table 3-141
- for data types 1-10
- removing a synonym 3-63
- rules for owner naming 1-8

## Syntax conventions, SQL statements Preface-7

## System catalogs

- creating 3-26
- definition of 1-6
- deleting when drop database 3-60
- described B-3
- querying when MODE ANSI 1-9
- systcolauth B-5

## System catalogs (cont)

- syscolumns 1-55, B-4
- sysconstraints B-8
- sysdepend B-6
- sysindexes B-4
- sys synonyms B-6
- sysstable B-8
- sysstauth B-5
- sysstables B-3
- sysusers B-7
- sysviews 1-55, B-7

## System failure, and table recovery 3-99

### Table

- access by ROWID 1-60
- access privileges 1-44
- adding a column 3-14
- adding a row F-8
- alias for table name 3-141
- altering 3-14
- creating an alternate name (synonym) 3-32
- creating a temporary table 3-165
- creating audit trail for 3-24
- creating index for 3-29
- definition of 1-6
- deleting a row 3-53
- dropping audit trail file for 3-59
- granting access to 3-79
- guidelines for indexing 1-46
- how to recover 1-36, 3-99
- joining columns 3-152
- locking 3-87, F-18
- modifying a column 3-120
- modifying a row 3-14
- naming conventions 1-7, 3-37
- not logging temporary 1-32, 3-35, 3-165
- outer join 1-59
- removing 3-65
- removing a column 3-14
- removing a synonym 3-63
- renaming 3-103
- revoking access to 3-105
- rules for owner naming 1-8
- structure in stores database E-4
- unlocking 3-119

- TEMP keyword, CREATE TABLE statement 3-34

- TEXT data type Preface-14

- TODAY function, definition of 1-61
- TODAY keyword, in SELECT statement 3-145

### Transaction

- and data integrity 1-36
- and MODE ANSI databases 1-33
- committing modifications 3-23
- compared to audit trail 1-36
- cycle 1-32
- definition of 1-31
- log 1-32, 3-27
- log file maintenance 1-34
- logging temporary tables 1-32
- pathname of log file 3-116
- recovering transactions 3-109
- single for INSERT statement 3-85
- starting 3-18
- undoing modifications 3-108

### Transactions

- and PUT statement 3-98
- and row-level locking 1-41
- and UPDATE statement 3-122
- definition of 1-32
- when to use 1-36

- Transfer of database files F-8

- Truncation of data F-13

### Type conversion

- for ACE and PERFORM 6-18
- functions in C program structure 6-18

### typedef

- dec\_t shown 4-27
- dtime\_t shown 4-70
- intrvl\_t shown 4-70

- UNION operator, SELECT statement 3-167

- UNIQUE constraint F-12

- UNIQUE CONSTRAINT, and DROP INDEX statement 3-62

### UNIQUE keyword

- ALTER TABLE statement 3-14
- and modifying database through view 1-56
- CREATE INDEX statement 3-29
- CREATE TABLE statement 3-34
- SELECT clause 3-134

- UNITS keyword, replacing INTERVAL value H-16

- UNIX permissions F-12

- unload program shown 2-40



- UNLOAD statement F-9
- UNLOCK TABLE statement
  - definition of 1-37
  - syntax and notes 3-119
- UPDATE keyword, GRANT statement 3-80
- UPDATE statement
  - data conversion in 3-122
  - definition of 1-15
  - guidelines for using 3-121
  - Informix extensions to 3-9
  - SET keyword 3-120
  - subqueries 3-121
  - syntax and notes 3-120
  - updating through a view 1-56
  - using DATETIME or INTERVAL values H-18
  - WHERE clause 3-121
  - WHERE CURRENT OF clause 1-22
  - with NULL values 1-53
- UPDATE STATISTICS statement
  - definition of 1-15
  - FOR TABLE clause 3-124
  - syntax and notes 3-124
- USER function, definition of 1-61
- USER keyword,
  - in SELECT statement 3-145
- User status and privileges 1-44
- USING DESCRIPTOR keywords
  - in EXECUTE statement 3-68
  - in FETCH statement 3-73
  - in OPEN statement 3-89
  - in PUT statement 3-96
- Utility programs
  - bcheck F-4
  - dbexport F-33
  - dbimport F-36
  - dbload F-8
  - dbschema F-21
  - dbupdate F-25
  - sqlconv F-27
- Values
  - defaults for columns 1-49
  - definition of NULL 1-49
  - fetching DATETIME 4-75
  - fetching INTERVAL 4-75
  - for DATETIME H-4
  - for DATETIME character strings H-11

- Values (cont)
  - for INTERVAL H-9
  - for INTERVAL character strings H-11
  - manipulating for DATETIME H-13
  - manipulating for INTERVAL H-15
  - multiplying or dividing INTERVAL H-17
  - no NULLs in dbupdate utility F-26
  - NULL in Boolean expressions 1-51
  - NULL in expressions 1-50
  - NULL in GROUP BY clause 1-53
  - NULL in ORDER BY clause 1-52
  - NULL in WHERE clause 1-51
  - NULL with INSERT statement 1-53
  - NULL with UPDATE statement 1-53
  - operations using CURRENT H-16
  - operations using UNITS H-16
  - returned to ACE 6-18
  - returned to PERFORM 6-18
  - storing DATETIME 4-76
  - storing INTERVAL 4-76
- VALUES keyword
  - dbload command file F-11
- VALUES keyword,
  - INSERT statement 3-83
- VARCHAR data type Preface-14
- Variable
  - conversion of data types 4-8
  - converting DECIMAL columns to FLOAT and DOUBLE 4-12
  - database versus host 4-8
  - data types for host 4-6
  - environment, how to set C-4, Preface-9
  - host 2-8
  - host as pointer to CHAR 4-10
  - indicator 2-14
  - rules for DECIMAL type 4-9
- Vertical ( | ) bar, field separator in files F-15
- View
  - access privileges 1-57
  - alternate name (synonym) 3-32
  - characteristics 3-44
  - creating 1-55, 3-43
  - data constraints on 1-58
  - definition and uses 1-54
  - deleting 1-55, 3-67
  - deleting when remove table 3-65

## View (cont)

- limitations 1-54
- modifying the database through 1-56
- querying the database through 1-55
- rules for owner naming 1-8
- virtual column 3-44
- with duplicate rows 1-57

## Wait for lock

- definition of 1-43
- SET LOCK MODE statement 3-114

## WAIT keyword, in SET LOCK MODE statement 3-114

## Weekday function, in a SELECT statement 3-178

## WHENEVER statement

- SQLERROR keyword 3-126
- SQLWARNING keyword 3-126
- syntax and notes 3-126

## WHERE clause

- ALL keyword 3-156
- ANY keyword 3-156
- BETWEEN keyword 3-144
- comparison condition 3-142
- current function 3-180
- date function 3-174
- day function 3-175
- ESCAPE keyword 3-146, 3-148
- EXISTS keyword 3-156
- extend function 3-182
- IN keyword 3-145, 3-156
- IS NULL keywords 3-150
- joining columns in 3-152
- LENGTH keyword in 3-173
- LIKE keyword 3-146
- MATCHES keyword 3-148
- mdy function 3-176
- month function 3-177
- NULL values 3-150
- pattern matching in 3-148
- search conditions 3-130, 3-142
- SELECT statement 3-142
- SOME keyword 3-156
- syntax and notes 3-142
- weekday function 3-178
- with a subquery 3-155
- with DELETE 3-53
- with NULL values 1-51
- with relational operators 3-143
- with UPDATE 3-121

## WHERE clause (cont)

- year function 3-179

## WHERE CURRENT OF clause

- DELETE statement 3-54

- UPDATE statement 3-121

## WITH CHECK OPTION clause, CREATE VIEW statement 3-43

## WITH GRANT OPTION clause, GRANT statement 3-80

## WITH HOLD cursors 1-17

## WITH HOLD keywords, DECLARE statement 3-48

## WITH LOG IN keywords

- CREATE DATABASE statement 1-32, 3-26

- START DATABASE statement 3-116

## WITH NO LOG keywords

- CREATE TABLE statement 3-35
- SELECT INTO TEMP statement 3-165

## Year function, in a SELECT statement 3-179

## YEAR keyword, DATETIME qualifier F-9

